Algorithmen und Datenstrukturen | AlgDat

Zusammenfassung

1. BINÄRE SUCH-BÄUME (BINARY SEARCH TREES, BST)

1.1. 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***)

1.1.1. Geordnete Multimaps

Die Keys folgen einer vollständigen Ordnungsrelation, das heisst, dass die Keys der Grösse nach sortiert sind ($k_m \leq k_n$). 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)*

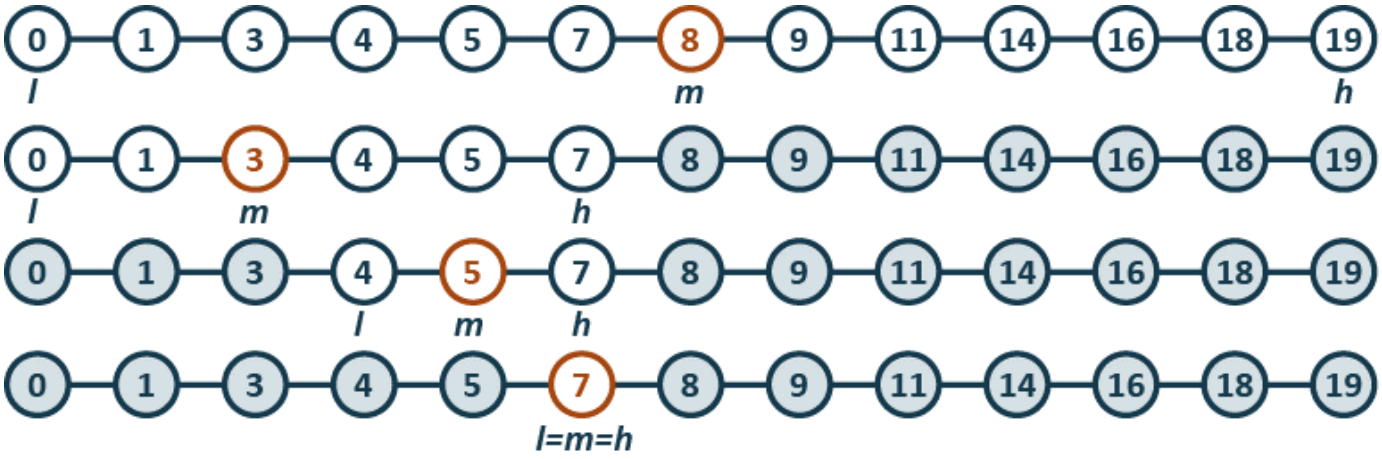
1.1.2. Suchtabelle

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.

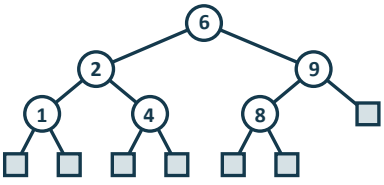
Binäre Suche (find(k))

Bei jedem Schritt wird die Anzahl der Kandidaten halbiert, terminiert nach $O(\log n)$ Schritten. ***Bsp: find(7):***

Gestartet wird in der Mitte des Arrays (m). Da $7 < 8$ werden nur die Entries auf der linken Seite von 8 angeschaut, das Ende des Bereichs (h) wird also auf 7 gesetzt. Nun wird die neue Mitte (3) bestimmt und wieder mit 7 verglichen. Wiederhole die Schritte, bis $l = m = h$.



1.2. BINÄRER SUCH-BAUM



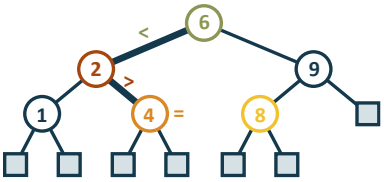
Binärer Baum, welcher Key in seinen internen Knoten speichert. Knoten (v) haben ***maximal 2 Kinder***, wobei einer davon im linken Teilbaum (u) und der andere im rechten Teilbaum (w) ist.
Bei den drei Knoten u, v und w gilt: $\text{key(u)} \leq \text{key(v)} \leq \text{key(w)}$

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

1.2.1. Suche nach Key k

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

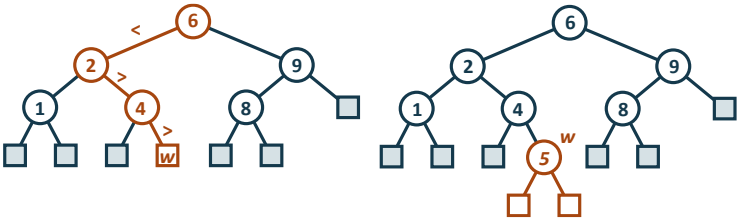
```
Algorithm TreeSearch(k,v)          //k = value to find, v = node to compare
 if T.isExternal(v)                //Node is Leaf, key not
   return v                        //found, return v (key(8))
 if k < key(v)                     //Ex: key(6)
   return TreeSearch(k, T.left(v))
 else if k > key(v)                //Ex: key(2)
   return TreeSearch(k, T.right(v))
 else if k = key(v)                //Ex: key(4)
   return v
```



1.2.2. Einfügen des Key k

***k einfügen*, falls noch nicht vorhanden**

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

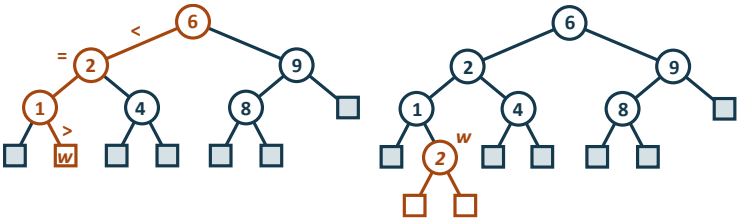


***k einfügen*, falls schon vorhanden (map)**

Value bei k wird ersetzt

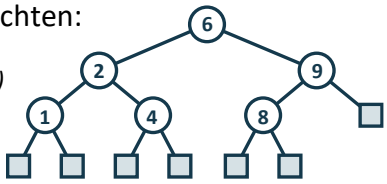
***k einfügen*, falls schon vorhanden (multimap)**

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



1.2.3. Löschen des Key k

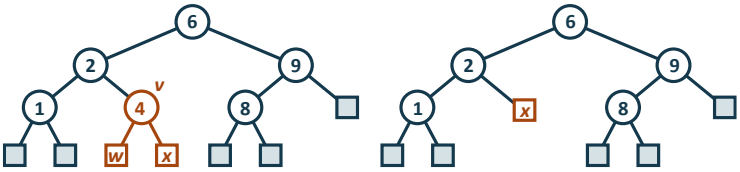
Zuerst Suche nach Key k. Anschliessend gibt es ***drei verschiedene Fälle*** zu beachten:



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

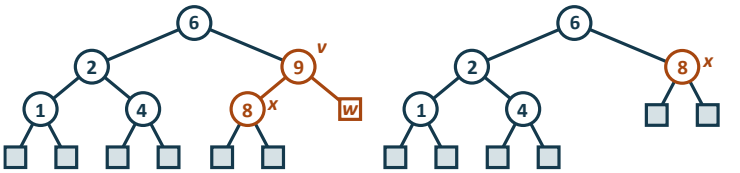
1. Löschen mit zwei Blatt-Kinder (Knoten 4)

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



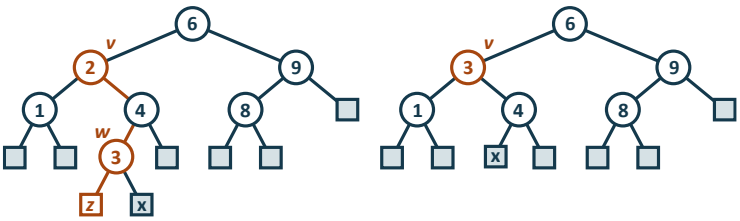
2. Löschen mit einem Blatt-Kind (Knoten 9)

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



3. Löschen ohne Blatt-Kinder (Knoten 2)

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

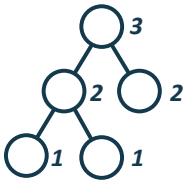


1.2.4. Gültige Suchpfade

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

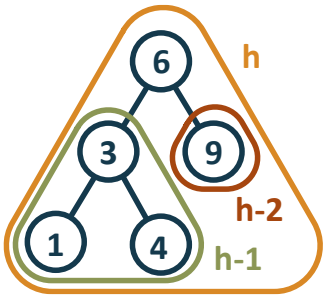
2. AVL BÄUME (ADELSON-VELSKY LANDIS)



Ein AVL-Baum ist ein binärer Such-Baum, bei dem für jeden internen Knoten v von T gilt: ***die Höhe der Kinder von v 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.

2.1. HÖHE EINES AVL BAUMES

Die Höhe eines AVL-Baumes ist $O(log(n))$



Begründung: Zuerst versuchen wir, die minimale Anzahl Knoten n, die für eine bestimmte Höhe h benötigt werden als Funktion darzustellen: $n(h)$. Sicher ist: $n(1) = 1\ Knoten$ und $n(2) = 2\ Knoten$. Für $\boldsymbol{h} \geq 3$ umfasst ein minimaler AVL-Baum der Höhe $\boldsymbol{h}$ die Wurzel, einen grösseren AVL-Unterbaum der Höhe $\boldsymbol{h-1}$ und einen zweiten, kleineren AVL-Unterbaum der Höhe $\boldsymbol{h-2}$.
Also gilt: $n(\boldsymbol{h}) = 1 + n(\boldsymbol{h-1}) + n(\boldsymbol{h-2})$

Da $n(h-1) > n(h-2)$ können wir die obige Formel umschreiben als $n(h) > n(\boldsymbol{h-2}) + n(\boldsymbol{h-2}) \rightarrow n(h) > 2n(h-2)$ *(das 1 kann weggelassen werden, weil* $\mathrm{n(h)}$ *auch ohne das 1 grösser ist)*

Daraus folgt, dass $n(\boldsymbol{h})$ in diesem Fall grösser sein muss als $2\mathrm{n(h-2)}$

Rekursiv gilt: $n(\boldsymbol{h-2}) = 1 + n(h-3) + n(h-4)$, was grösser sein muss als $2n(n-4)$, woraus folgt, dass $n(\boldsymbol{h}) > \boldsymbol{4n(h-4)}$

…

Daraus lässt sich verallgemeinern, dass $\mathrm{n(h)} > 2^{\mathrm{i}}\mathrm{n(h-2i)}$.

Da $\mathrm{h-2i} = 1$, können wir $\mathrm{i} = \mathrm{h}/2 - 1$ setzen. Daraus folgt:

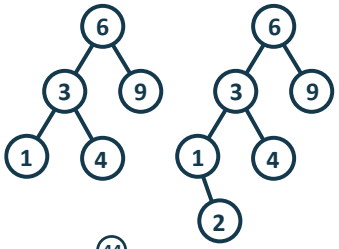
$n(h) > 2^{h/2-1}n(1) \Rightarrow \log n(h) > \log(2^{h/2-1}) \Rightarrow \log n(h) > h/2 - 1 \Rightarrow h < 2 * \log n(h) + 2$
Was bedeutet, dass die Höhe eines AVL-Baumes tatsächlich $O(\log(n))$ ist.

2.2. BALANCIERUNGSFAKTOR

Ein AVL-Baum ist ***balanciert*** (b). Das heisst, die Höhe des linken und des rechten Teilbaums eines Knoten v unterscheidet sich maximal um eins.
$b(k) = Höhe(links) - Höhe(rechts)$
$b(k)$ kann also nur entweder $-1, 0$ oder $+1$ sein.
Die untersten Knoten (Blätter) haben immer einen Balancierungsfaktor von 0. Bei den anderen Knoten wird der linke Teilbaum minus der rechte Teilbaum gerechnet.

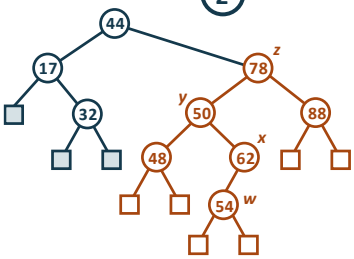
- Linker Teilbaum um 1 höher: $+1$
- Rechter Teilbaum um 1 höher: -1
- Beide Teilbäume gleich hoch: 0

2.3. EINFÜGEN IN EINEN AVL BAUM, BALANCIERUNGSFAKTR



Nach dem ***Einfügen eines neuen Knotens*** kann $b(k)$ allerdings +2/-2 sein ($-2 \leq b(k) \leq 2$), 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***

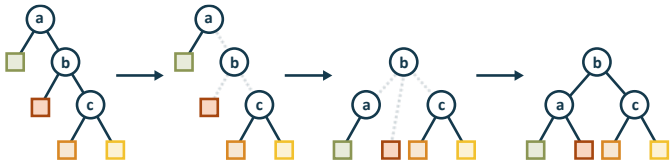
Der oberste Knoten mit dem zu grossen Balancierungsfaktor ist der z Knoten, das direkte Kind im längeren Teilbaum ist y und das direkte Kind von y im längeren Teilbaum ist x. Diese Knoten werden inorder mit a, b und c beschriftet.

2.3.1. Trinode Umstrukturierung

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

Linksrotation

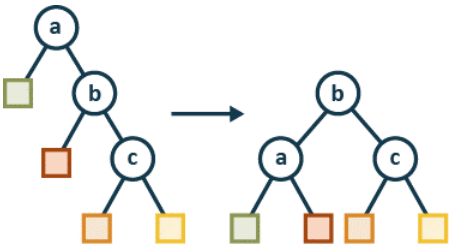
1. Alle Kanten von b trennen
2. b als root setzen
3. Kanten von b und linkem Kind verbinden



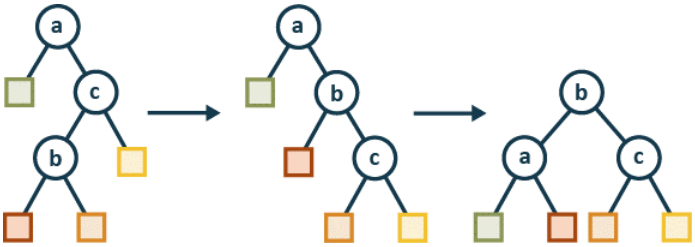
Rechtsrotation

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

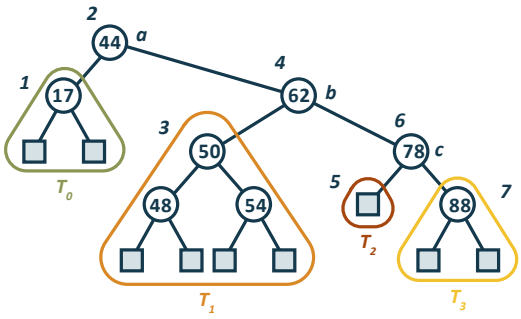
Fall 1: Einzel-Rotation *(Knoten sind inorder)*
(eine Links-Rotation um a*)*



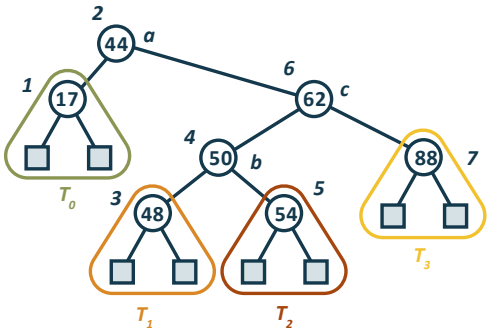
Fall 2: Doppelte Rotation *(Knoten sind nicht inorder)*
(eine Rechts-Rotation um c*, dann eine Links-Rotation um* a*)*



2.3.2. 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 z Knoten, das direkte Kind im längeren Teilbaum ist y und das direkte Kind von y im längeren Teilbaum ist x. Diese Knoten werden inorder mit a, b und c beschriftet.*** Die restlichen Kinder dieser Knoten werden von links nach rechts als ***Teilbäume T_0 … T_4*** gelabelt.



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

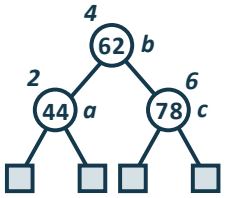
1	2	3	4	5	6	7

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

T_0		T_1		T_2		T_3
1	2	3	4	5	6	7

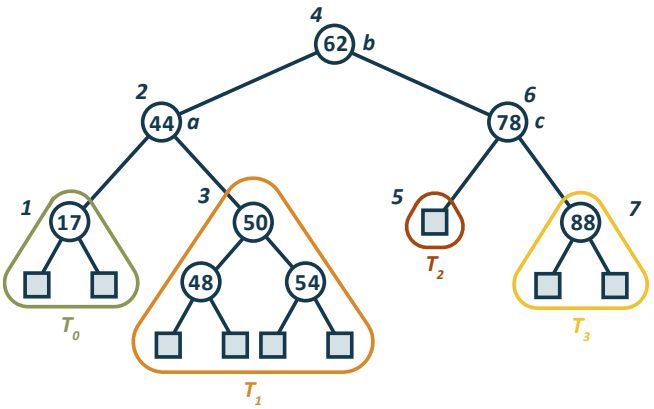
Jetzt stellen wir c, b ***und*** a in Inorder ***in die Positionen zwischen den Trees in das Array***.

T_0	$44_{(a)}$	T_1	$62_{(b)}$	T_2	$78_{(c)}$	T_3
1	2	3	4	5	6	7



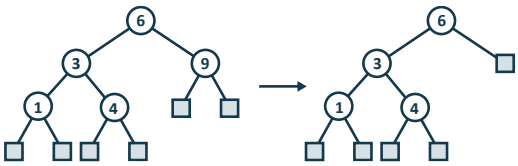
Nun ***bauen*** wir den ***Baum schrittweise wieder auf***. Wir beginnen mit b und setzen die anderen beiden Hauptknoten Element 2 (a) und Element 6 (c) als Kinder von 4 (b) in den Baum ein.

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

2.4. LÖSCHEN



Beginnt wie im binären Suchbaum, das heisst, der gelöschte Knoten wird ein leerer externer Knoten. Sein Eltern-Knoten "6" 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.

3. 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 v sind ***kleiner*** oder gleich wie der Key in v
- Keys gespeichert im ***rechten Teilbaum*** von v sind ***grösser*** oder gleich wie der Key in v

Eine ***Inorder-Traversierung*** retourniert die Keys in ***geordneter Folge***.

3.1. SUCHEN IN EINEM SPLAY-TREE

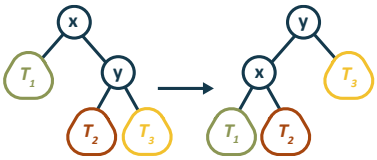
Suche funktioniert gleich wie in einem Binary Search Tree: Man geht den Baum abwärts bis zu einem gesuchten Entry oder einem externen Knoten.

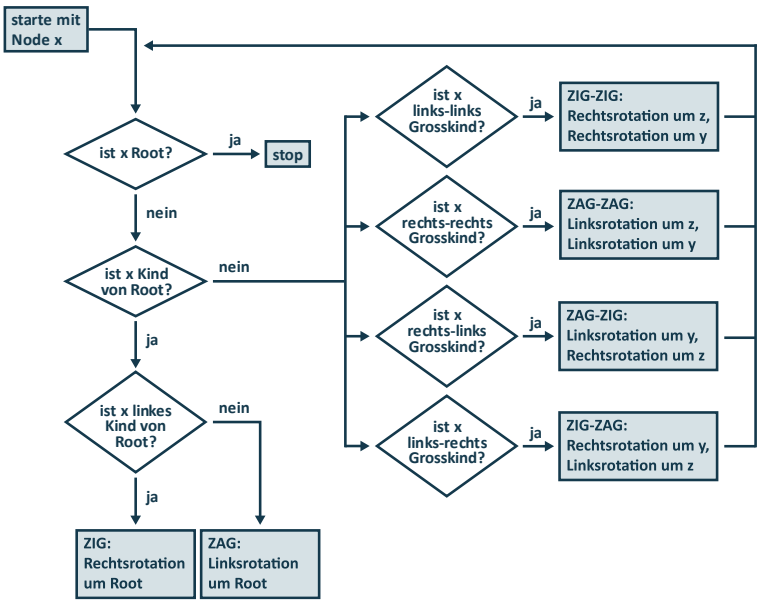
3.2. SPLAY

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

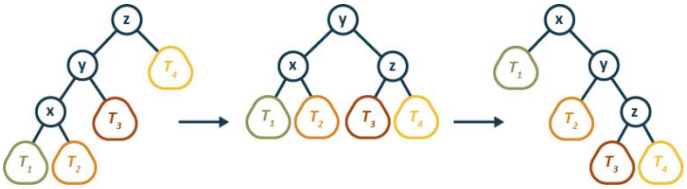
ZIG ***(Rechtsrotation)***
Macht das linke Kind x des Knoten y zu y's Eltern-Knoten, y wird zum rechten Kind von x

ZAG ***(Linksrotation)***
Macht das rechte Kind y des Knoten x zu x's Eltern-Knoten, x wird zum linken Kind von y

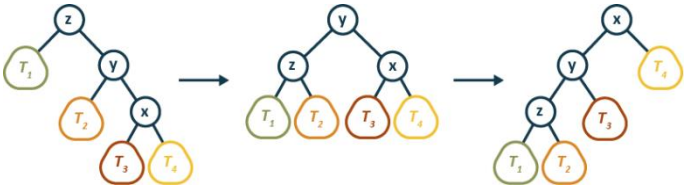




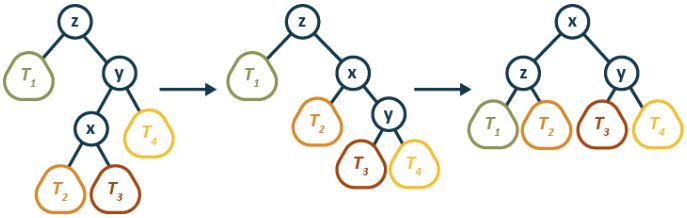
Zig-Zig ***(Rechts-Rechts)***



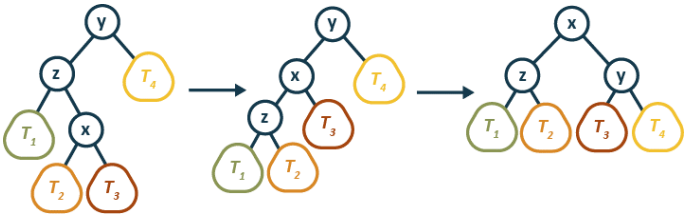
Zag-Zag ***(Links-Links)***



Zig-Zag ***(Rechts-Links)***

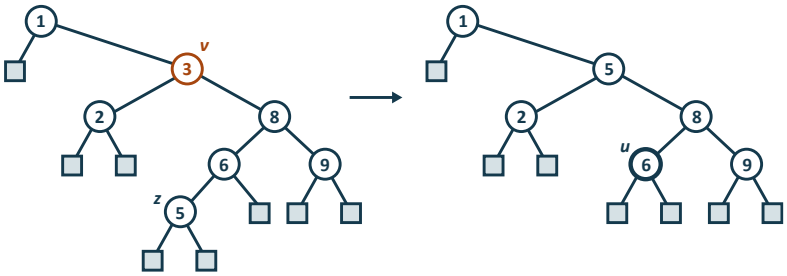


Zag-Zig ***(Links-Rechts)***

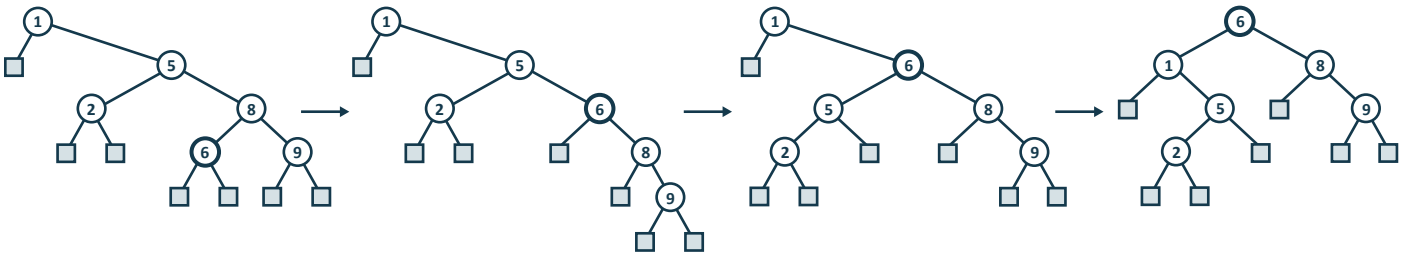


Beispiel für Löschen: remove(3)

Da v 2 interne Knoten hat, wird der Fall 3 angewendet *(siehe Löschen ohne Blatt-Kinder (Knoten 2))*: Ersetze v durch Inorder-Nachfolger z und lösche z. Das rechte Blatt von z wird an u angehängt, damit ist dieser der ***tiefste intern zugegriffene Knoten***, somit wird u 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*

4. 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.

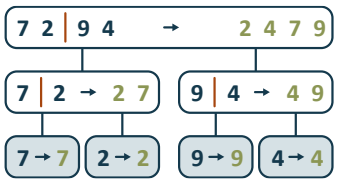
4.1. MERGE-SORT

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

- ***Divide:*** Input-Daten S in zwei getrennte Hälften S_1 und S_2 aufteilen
- ***Recur (Wiederhole):*** Rekursiv die Teilmengen S_1 und S_2 sortieren
- ***Conquer:*** S_1 und S_2 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 S, S_1 & S_2 idealerweise double-linked Lists, da diese vorne und hinten mit $O(1)$ zugreifen können.

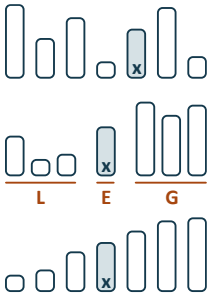
Die Ausführung eines Merge-Sort kann als ***binärer Baum*** dargestellt werden. Jeder Knoten repräsentiert einen rekursiven Aufruf des Merge-Sort und enthält die ***unsortierte*** und die ***sortierte*** Sequenz. Die Wurzel entspricht dem initialen Aufruf. Die Blätter sind Aufrufe auf Teilsequenzen der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



Beispiel: $S = [7, 2, 9, 4]$ wird so lange rekursiv in der Mitte aufgeteilt, bis $S_1 = [7]$ ist, dann wird S_1 zurückgegeben. Ist dann $S_2 = [2]$ werden die beiden vereint: Die Elemente werden verglichen, der kleinere Wert (2) zuhinterst in S eingefügt und aus S_1 entfernt. Da S_1 nun leer ist, werden alle Elemente aus S_2 direkt in S eingefügt. Das Ganze wiederholt sich mit $S_1 = [2, 7]$ und $S_2 = [4, 9]$.

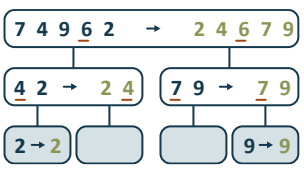
4.2. QUICK-SORT

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 x *(genannt Pivot)* und Aufteilung von S in L *(less, Elemente kleiner als* x*)*, E *(equal, Elemente gleich* x*)* und G *(greater, Elemente grösser als* x*)*
- ***Recur:*** sortiere L und G
- ***Conquer:*** vereine L, E und G 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: $S = [7, 4, 9, 6, 2]$. 6 wird als Pivot ausgewählt. Nun werden alle anderen Elemente mit 6 verglichen, aus S entfernt und entweder in $L = [4, 2]$, $E = [6]$ oder $G = [7, 9]$ eingeteilt. L & G werden rekursiv weiter aufgeteilt, bis die Rekursion mit 1 Element in der Liste beendet wird. Die Listen werden

dann wieder zu einer vereinigt, indem L, E & G nacheinander eingefügt werden. Diese Liste wird ein Rekursionslevel hochgegeben, bis die Originalliste sortiert ist.

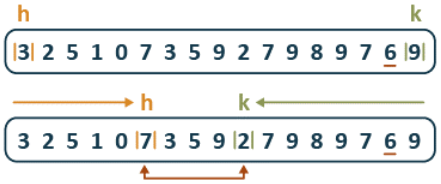
Anzahl Vergleiche ist auf jeder Höhe die Anzahl der Elemente – Anzahl der Pivots auf dieser Stufe.

4.2.1. In-Place Quick-Sort

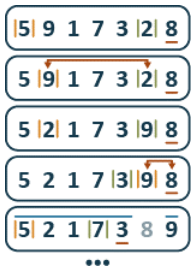
Quick-Sort kann auch In-Place durchgeführt werden, also im selben Array, ohne dass die Elemente in weitere Arrays kopiert werden müssen. Dadurch wird vor allem bei grossen Inputs viel RAM gespart.

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

- Die Elemente $\leq$ ***Pivot*** einen Index kleiner als h haben
- Die Elemente $\geq$ ***Pivot*** einen Index grösser als k haben



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



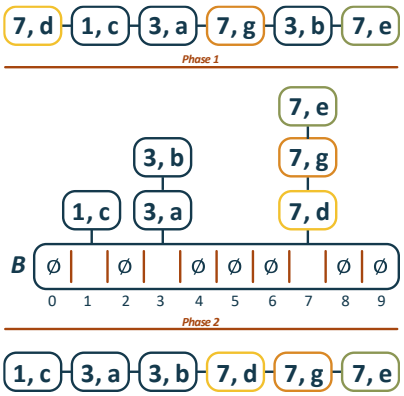
4.3. UNTERE GRENZE DER VERGLEICHSBASIERTEN SORTIERUNG (LOWER BOUND)

Jeder Vergleichs-basierte Sortier-Algorithmus muss mindestens eine Laufzeit von $O(\log n!) \approx O(n * \log n)$ haben.

Erklärung: Jeder vergleichsbasierte Algorithmus kann als Binary Tree dargestellt werden (Entscheidungsbaum). In diesem Entscheidungsbaum gibt es $n! + 1$ Knoten. Je nachdem wie ein Vergleich von zwei Elementen ausfällt, ändert sich der Pfad durch den Entscheidungsbaum (z.B. $a < b$ führt zu einem anderen Pfad als $a > b$). Da ein Entscheidungsbaum immer ein balancierter Binary Tree ist, ist dessen Höhe $log(n)$, eine Pfadtraversierung ist also $O(\log(n!))$. Mit der Stirling-Approximation kann dies als etwa $O(n * \log n)$ beschrieben werden.

5. 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 $O(n)$.



5.1. BUCKET-SORT

Sei S eine Sequenz von n (Key, Element) Entries mit Keys im Bereich von $[0, N-1]$. Bei Bucket-Sort werden die Entries anhand der Keys in einem Hilfsarray B mit Grösse N abgelegt. Hinter jedem Index von B 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 B herausgenommen werden.

- ***Phase 1:*** Sequenz S leeren durch Verschieben jedes Entry (k, o) in sein Bucket $B[k]$
- ***Phase 2:*** Für $i = 0, \dots, N-1$, verschiebe die Entries des Buckets $B[i]$ an das Ende der Sequenz S

5.1.1. 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

5.2. LEXIKOGRAPHISCHE ORDNUNG (STABLE SORT)

In einem Tupel werden die ***Werte der gleichen Dimension*** der Reihe nach ***verglichen***. Die Lexikographische Sortierung sortiert eine Sequenz von d-Tupeln in lexikographischer Ordnung, indem d-mal ein stabiler Sortier-Algorithmus *(«stableSort»)*, welcher den Comparator C 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. $[2,1,4] < [2,4,6]$

Mathematische Notation für Sortierung von zwei Tupeln $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$:

$p_1 < p_2 \mid x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2)$

$p_1 = p_2 \mid x_1 = x_2 \wedge y_1 = y_2$

$p_1 > p_2 \mid \neg(p_1 < p_2) \wedge \neg(p_1 = p_2)$

5.3. 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 $[0, N-1]$ in jeder Dimension i. Es wird wie bei der lexikographischen Sortierung jede Dimension i der Tupel von hinten via Bucket Sort sortiert. Diesem wird neben dem Input S noch die maximale Keygrösse N sowie die aktuelle Dimension i übergeben.

5.3.1. Binärer Radix Sort

Gegeben sei eine Sequenz von n b-Bit Integers $x = x_{b-1} \dots x_1 x_0$. Jedes Element wird als b-Tupel von Integern im Bereich $\{0,1\}$ dargestellt. Darauf wendet man den Radix-Sort mit $N = 2$ an *(da nur 0 und 1 als Keys möglich sind)*. Diese Anwendung des Radix-Sort-Algorithmus läuft in $O(b * n)$ Zeit. ***Beispiel***: Eine Sequenz von 32-Bit Integers kann in ***linearer Zeit*** sortiert werden, nämlich $O(32 * n)$.

6. ZUSAMMENFASSUNG DER SORTIER-ALGORITHMEN

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

Algorithmus	*Zeitverhalten*	*Bemerkungen*
selection-sort	$O(n^2)$	- Langsam & instabil - in-place - für kleine Data-Sets $(< 1\text{K})$ - Profitiert nicht von einem vorsortierten Input
insertion-sort	$O(n^2)$	- Langsam & stabil - in-place - für kleine Data-Sets $(< 1\text{K})$ - Für kleine Inputs besser als selection-sort
Quick-sort	$O(n * \log n)$ *durchschnittlich, wegen zufälligem Pivot*	- Kann in-place implementiert werden (nicht stabil) - meist instabil (implementationsabhängig) - schnellster (gut für grosse Inputs)
heap-sort	$O(n * \log n)$	- schnell & instabil - in-place - für grosse Data-Sets $(1\text{K} - 1\text{M})$
merge-sort	$O(n * \log n)$	- schnell & stabil - sequenzieller Datenzugriff *(es müssen jeweils nur 2 Elemente gleichzeitig in den RAM geladen werden)* - für riesige Data-Sets $(> 1\text{M})$
Bucket-sort	$O(n + N)$	- Benötigt Integer als Keys - Stabil
Lexikographische Sortierung	$O\big(d * T(n)\big)$, *wobei* $\mathrm{T(n)}$ *der einzelnen Sortierung des anderen Algos ist*	- Benötigt Integer als Keys - In-place - Stabil
Radix-Sort	$O\big(d * (n + N)\big)$	- Benötigt Integer oder Bits als Keys - In-place - Stabil

7. 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 Σ ist ein Set von möglichen Zeichen für eine Familie von Strings. Beispiele für Alphabete: ASCII, Unicode, $\{0,1\}$, $\{A, C, G, T\}$.

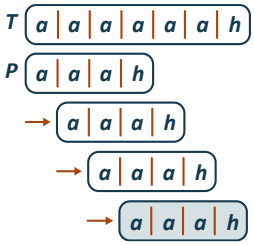
Sei P *(Pattern)* ein String der Länge m. Ein Substring $\mathrm{P[i..j]}$ von P ist die ***Subsequenz*** von P, bestehend aus den Zeichen mit Index zwischen und ***inklusiv*** i und j.

Spezielle Substrings

- Ein ***Präfix*** von P ist ein Substring vom Typ $P[0..i]$ *alle Zeichen bis* i
- Ein ***Suffix*** von P ist ein Substring vom Typ $P[i..m-1]$ *alle Zeichen nach* i

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

7.1. BRUTE-FORCE PATTERN MATCHING



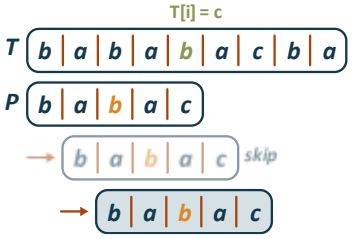
Der Brute-Force Pattern Matching Algorithmus vergleicht das Pattern P mit dem Text T für ***jede mögliche Position*** von P relativ zu T, bis entweder ***eine Übereinstimmung*** gefunden wurde oder ***alle möglichen Platzierungen*** des Patterns ausprobiert wurden. Sind 2 einfache Loops für T & P, bei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

Worst-Case Beispiel: $T = aaa \dots ah, P = aaah$. Kann in Bild-Analysen und DNA-Sequenzanalysen eintreten, in sprachlichen Texten eher nicht.

7.2. BOYER-MOORE ALGORITHMUS

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

- ***«Looking-Glass:*** Vergleiche P mit einer Subsequenz von T. Startet am Anfang des Strings, die einzelnen Charakters werden mit dem Pattern aber von hinten nach vorne verglichen.
- ***«Character-Jump»:*** falls bei $T[i] = c$ keine Übereinstimmung: falls P das Zeichen c enthält, verschiebe P bis das letzte Auftreten von c in P mit $T[i]$ übereinstimmt. Ist das hinterste Auftreten vor der aktuellen Position, wird stattdessen um 1 geschoben.
  Ansonsten, verschiebe P, bis $P[0]$ mit $T[i+1]$ übereinstimmt *(das ganze Pattern vor* T[i] *schieben)*. Damit lassen sich Positionen, welche ohnehin keinen Match geben würden, überspringen.



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

7.2.1. Last Occurrence Funktion

Der Boyer-Moore Algorithmus erstellt eine Last Occurrence Funktion, während er den Text T durchsucht. In dieser Tabelle ist jeder Buchstabe nur maximal 1x vorhanden.

Beispiel: $\Sigma = \{a, b, c, d\},\ P = a_o b_1 a_2 c_3 a_4 b_5$ (Index der Zahl ist Tiefgestellt) ergibt folgende Tabelle:

c	***a***	***b***	***c***	***d***
$L(c)$ *letztes Vorkommen des Zeichens*	**4**	**5**	**3**	-1 *(kommt nicht vor)*

Diese Funktion $L(c)$ lässt sich als ein Array darstellen, dessen Indices durch numerische Werte des Alphabets gegeben sind. Diese Funktion lässt sich in $O(m + s)$ berechnen, wobei m die Länge von P und s die Anzahl Zeichen in Σ ist.

7.2.1.1. Berechnung der Verschiebung

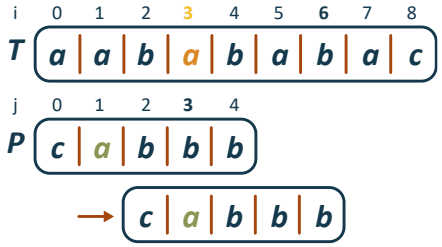
Fall 1: Zeichen kommt im Pattern vor

Das Zeichen $T[i]$ *(Zeichen im Text an Stelle* i*)* ***kommt im Pattern vor*** $(L(T[i]) > -1)$. In diesem Fall müssen wir bis zum letzten Auftreten des Zeichens $T[i]$ im Pattern P verschieben. Die Berechnung des neuen i's *Index letztes Zeichen von P = i + Patternlänge – (Index letztes Vorkommen + 1)*: $i = \boldsymbol{i} + \boldsymbol{m} - (\boldsymbol{L}(\boldsymbol{T}[\boldsymbol{i}]) + 1)$

Beispiel im Bild:

c	**a**	**b**	**c**	**d**
$L(c)$	**1**	4	0	-1

$\boldsymbol{i = 3, T[i] = a, m = 5} \Rightarrow i = \boldsymbol{3} + \boldsymbol{5} - (\boldsymbol{1} + 1) = 8 - 2 = \boldsymbol{6}$



Das nächste Zeichen, welches überprüft wird, ist das an Index 6.
Das letzte Vorkommen vom Zeichen $T[i] = a$ im Pattern P wurde damit an die Stelle $i = 3$ verschoben.

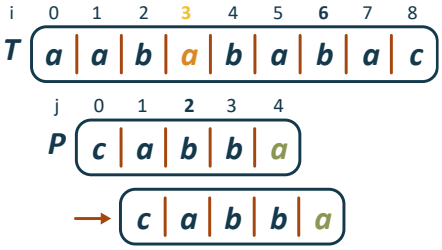
Fall 2: Zeichen kommt vor, letztes Vorkommen ist aber bereits vorbei

Das Zeichen $T[i]$ ***kommt im Pattern vor*** $(L(T[i]) > -1)$, ist aber bereits vorbei. In diesem Fall wird das Pattern einfach ***um eine Stelle nach vorne*** geschoben. Die Berechnung des neuen i's: $i = i + m - j$ *j = Index von P[i]*

Beispiel im Bild:

c	**a**	**b**	**c**	**d**
$L(c)$	4	3	0	-1

$\boldsymbol{i = 3}, P[i] = b, \boldsymbol{j = 2}, \boldsymbol{m = 5} \Rightarrow i = \boldsymbol{3} + \boldsymbol{5} - \boldsymbol{2} = 8 - 2 = \mathbf{6}$



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

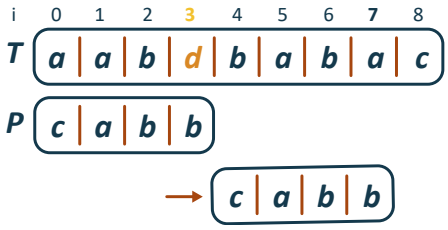
Fall 3: Das Zeichen kommt nicht im Pattern vor

Das Zeichen $T[i]$ ***kommt im Pattern nicht vor*** $(L(T[i] = -1)$. In diesem Fall wird das ganze Pattern vor das Zeichen $T[i]$ geschoben, da $L(T[i])$ -1 zurückgibt, welches durch das $+1$ wieder ausgeglichen wird.

Beispiel im Bild:

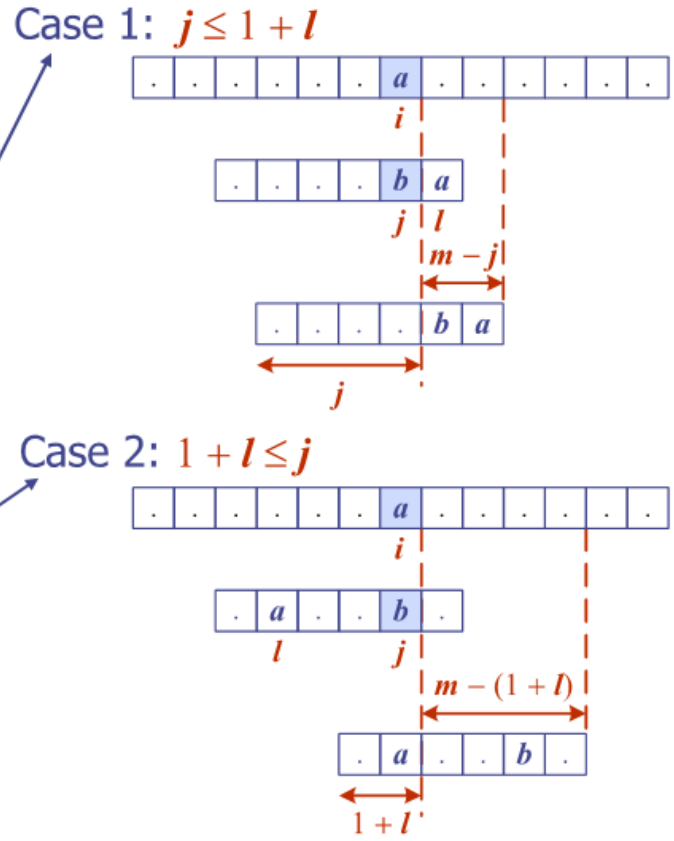
c	**a**	**b**	**c**	**d**
$L(c)$	1	3	0	$\boldsymbol{-1}$

$\boldsymbol{i = 3, T[i] = d, m = 4} \Rightarrow i = \boldsymbol{3} + \boldsymbol{4} - (\boldsymbol{-1} + 1) = 7 - 0 = \mathbf{7}$



Im Pseudocode wird die Fallunterscheidung mit $i = i + m -$ $\min(j, 1 + L(T[i]))$ beschrieben, wobei j Fall 2 und $L(T[i])$ Fall 1 & 3 entspricht.

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ)
    i ← m − 1        {actual T-index}
    j ← m − 1        {actual P-index}
    repeat
        if T[i] = P[j]
            if j = 0
                return i  { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m − min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1 { no match }
```



Boyer-Moore BM:

a	d	b	a	a	c	a	a	b	e	d	a	c	e	d	b	c	c	e	d	e
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
d	a	e	d	a		<- Spezialfall: statt einer negativen Verschiebung nach links: eine Stelle nach rechts														
	d	a	e	d	a		<- optimaler Fall: mismatched Character 'c' kommt im Pattern überhaupt nicht vor: Shift um das ganze Pattern													
						d	a	e	d	a		<- Ausrichtung auf mismatched Character 'd'								
							d	a	e	d	a		<- Shift ganzes Pattern hinter mismatched Character 'b'							
									d	a	e	d	a		<- Ausrichtung auf mismatched Character 'e'					
											d	a	e	d	a		<- optimaler Fall			
														d	a	e	d	a	i > n-1 : No Match	
																			: 11 Vergleiche	

c	a	b	c	d	e
$L(c)$	4	-1	-1	3	2

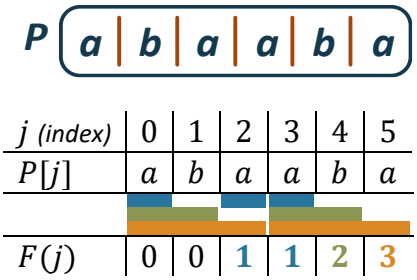
7.3. KMP ALGORITHMUS

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 $P[0..j]$ *(längstes Teilpattern von String-Beginn)* welches gleichzeitig Suffix von $P[1..j]$ *(längstes Teilpattern bei String-Ende)* ist. ***Beispiel***: $abaab$

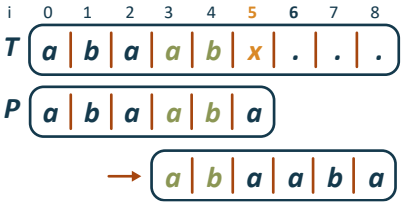
7.3.1. KMP Fehl-Funktion

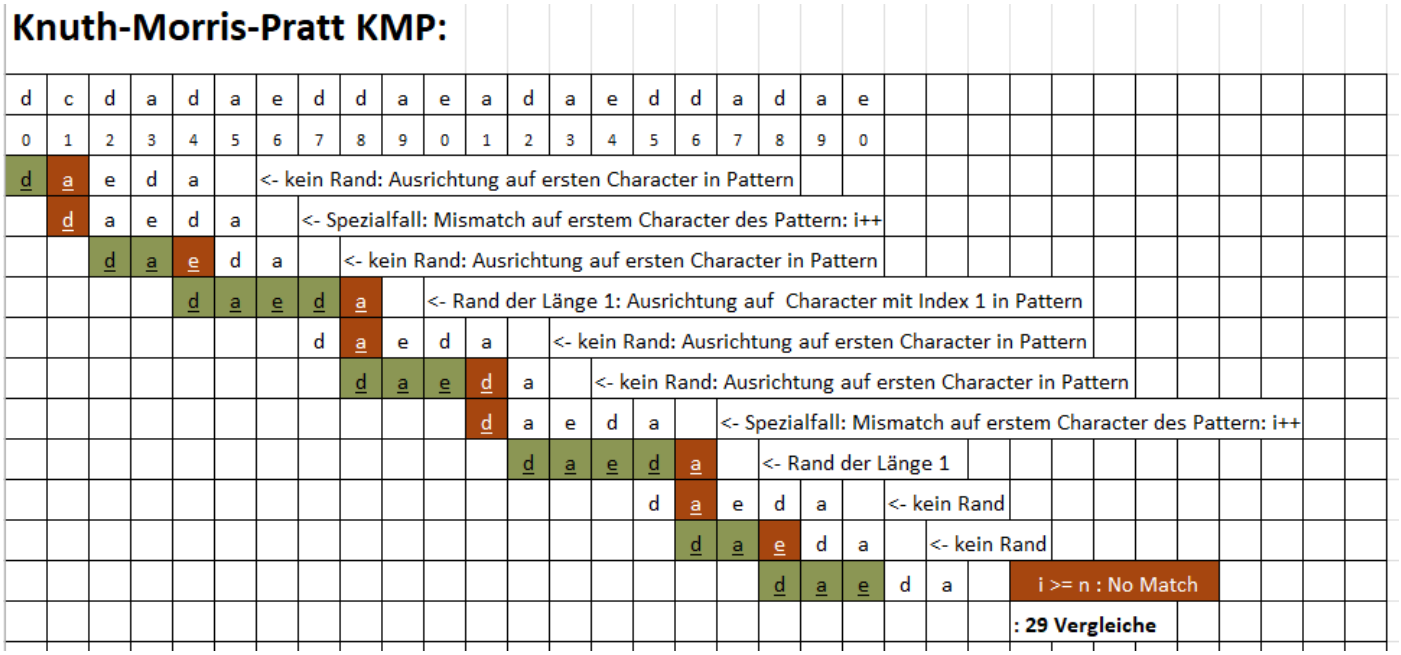
In einer Vorlaufsphase (Preprocessing) sucht der Algorithmus ***Übereinstimmungen von Präfixen des Musters im Muster selbst.*** Die Fehl-Funktion $F(j)$ ist definiert als die ***Grösse des längsten Präfixes*** von $P[0..j]$, sodass dieser auch Suffix von $P[1..j]$ ist. KMP modifiziert den Brute-Force-Algorithmus so, dass bei einer Differenz $P[j] \neq T[i]$ der Index j gesetzt wird mit $j \leftarrow F(j-1)$

Ränder-Länge:	**0**	**1**	**2**	**3**	**max. Länge $F(j)$**
$P[0] = a$	{}				0
$P[1] = ab$	{}				0
$P[2] = aba$	{}	a			1
$P[3] = abaa$	{}	a			1
$P[4] = abaab$	{}		ab		2
$P[5] = aba\,aba$	{}	a		aba	3



Bei $T[i = 5]$ steht im Text ein x, und im Pattern ein a. Das längste passende Suffix von P ist in diesem Fall ab (aba passt nicht, da hinten das x steht). Die Fehlerfunktion $F[j]$ von ab steht am Index 4, also wird das Pattern um $F[j-1] = F[4-1] = \mathbf{3}$ nach vorne geschoben.





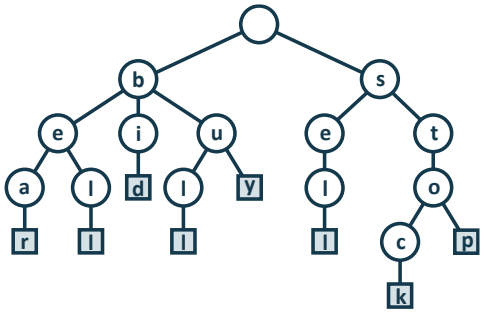
$P(j)$	d	a	e	d	a
$f(j)$	0	0	0	1	2

8. 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.

8.1. STANDARD TRIES

Ein Standard-Trie für eine Menge von Strings S 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 S beinhalten. Ist ***alphabetisch*** sortiert und ***case-sensitive***.



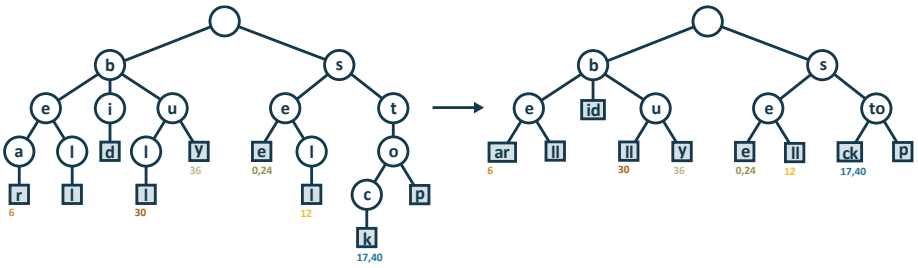
Beispiel eines Standard-Tries für die Menge von Strings
$S = \{$ bear, bell, bid, bull, buy, sell, stock, stop $\}$

8.2. 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

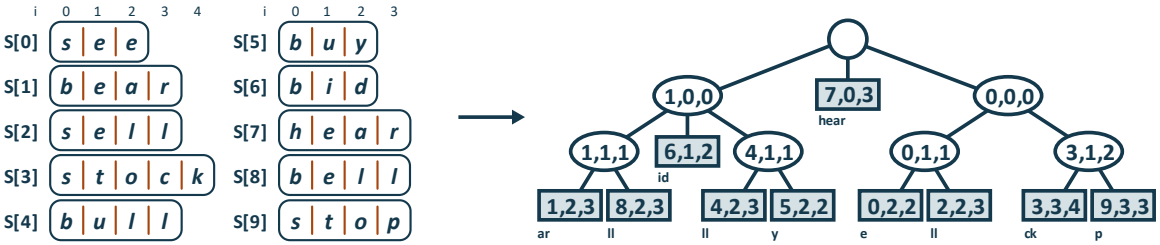
Ein ***komprimierter Trie*** wird von einem Standard-Trie hergeleitet durch Komprimierung von redundanten Knoten. Ein Knoten gilt als redundant, wenn es an ihm nur einen Pfad gibt.



8.2.1. Kompakte Trie-Repräsentation als Array

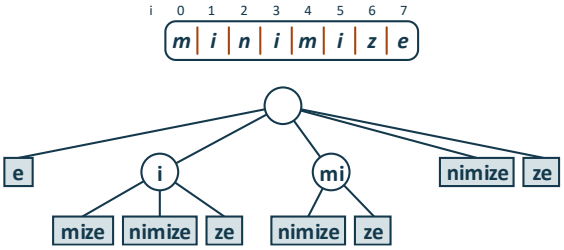
Eine ***kompaktere Repräsentation*** eines komprimierten Tries für ein Array von String wird erreicht, in dem Knoten die ***Indizes anstelle von Substrings*** speichert. Benötigt $O(s)$ Speicher, wobei s die Anzahl Strings im Array ist. Dient als eine Hilfs-Index-Struktur.

Erster Wert im Trie: Index s[i]*, zweiter Wert: Anfang des Substrings, dritter Wert: Ende des Substrings in* s[i]*, [x,0,0] = Ein Buchstabe*



8.3. SUFFIX TRIE

Der Suffix-Trie eines Strings X ist der komprimierte Trie von allen Suffixen von X. 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 S der Länge n

- Der Baum hat genau n Blätter.
- Mit Ausnahme der Wurzel hat jeder interne Knoten mindestens zwei Kinder.
- Jeder Knoten ist mit einem nicht-leeren Substring von S 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 i gefunden wurden, ergibt Suffix $S[i..n]$, für $i = 1..n$

9. ZUSAMMENFASSUNG DER PATTERN-MATCHING-ALGORITHMEN

Algorithmus	*Zeitverhalten*	*Bemerkungen*
Brute-Force	$O(n * m)$	- Langsamste, aber simpelste Methode
Boyer-Moore	$O(n * (m + s))$	- Bei Textanalysen signifikant schneller als Brute-Force
KMP	$O(m + n)$	- Geeignet für Strings mit vielen gleichen Buchstaben-Pattern
Trie	$O(dm)$ d = *Grösse des Alphabets*	- Gut, um Wörter zu finden, die mit dem gleichen Buchstaben beginnen - Suchzeit ist unabhängig von Anzahl Wörter

10. DYNAMISCHE PROGRAMMIERUNG

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

10.1. RUCKSACK-PROBLEM

Gegeben: n 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.

10.1.1. Brute-Force

Versuche alle möglichen Varianten. Betrachte nur jene Varianten, welche das maximale Gewicht nicht überschreiten. Nimm die beste Variante.
Laufzeit: Anzahl ist exponentiell $2^n \Rightarrow O(2^n)$, also sehr schlecht. *Die Reihenfolge des Durchprobierens spielt hier keine Rolle, ansonsten wäre die Laufzeit* $O(n!)$

10.1.2. Greedy

Nimm wiederholend den Gegenstand mit grösstem Verhältnis von Wert/Gewicht.
Beispiel: 4 Gegenstände, Rucksack-Kapazität: 5kg
Resultat: **4 *Fr.*** (3kg) + **1 *Fr.*** (1kg) = 5 $Fr.$ ⇒ 4kg mit einem totalen Wert von 5 $Fr.$ Das ist jedoch nicht optimal, besser wäre **1 *Fr.*** (1kg) + **5 *Fr.*** (4kg) = 6 $Fr.$ ⇒ 5kg mit einem totalen Wert von 6 $Fr.$ Also auch nicht optimal.

Wert	*Gewicht*	*W/G*
CHF 1	1kg	1.00
CHF 4	3kg	1.33
CHF 5	4kg	1.25
CHF 1	2kg	0.50

10.1.3. 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:
- 1..4 Gegenstände
- 1..5kg maximales Gewicht

Subproblem: Was ist der grösstmögliche Wert für die ersten beiden Gegenstände **1/1** und **4/3** bei einer Gewichtslimite von 3kg?
Lösung: grösstmöglicher Wert für dieses Subproblem: **4** mit Gegenstand **4/3**

↓ *Wert / kg, kg* →	1	2	3	4	5
CHF 1/1kg (a)	**1**	**1**	**1**	**1**	**1**
CHF 4/3kg (b)	**1**	**1**	**4**	**5** (4 + 1)	**5** (4 + 1)
CHF 5/4kg (c)	**1**	**1**	**4**	**5**	**6** (5 + 1)
CHF 1/2kg (d)	**1**	**1**	**4**	**5**	**6** (5 + 1)

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 b in den Rucksack gepackt wird. Mit 4 kg können bei a & b in den Rucksack gepackt werden → Wert 5. Am Ende haben wir a & d mit einem Gesamtwert von 6 im Rucksack.

Anschliessend Pfad (gewählte Gegenstände) bestimmen: Grösstmöglicher Wert ist $\mathbf{6}$.

- 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, c zu b). 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) ***abgezogen*** werden, neuer Wert: $5 - 4 = 1$
- Fahre mit dem nächsten ***Subproblem*** weiter: Gewicht ist neu 1kg, wo kommt 1 das erste Mal vor? Bei a & 1kg. a gehört ebenfalls in den Rucksack und das Feld gehört zum Pfad.

10.2. 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.* $\mathrm{j}, \mathrm{k}, \mathrm{l}, \mathrm{m}$*)*
- ***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)*

10.3. LÄNGSTE GEMEINSAME SUBSEQUENZ (LONGEST COMMON SUBSEQUENCE LCS)

Eine Subsequenz eines Character-Strings $x_0x_1x_2 \dots x_{n-1}$ ist ein String der Form $x_{i_1}x_{i_2} \dots x_{i_k}$, wobei $i_j < i_{j+1}$. Die ***Reihenfolge ist gleich*** wie beim Hauptstring, die ***Buchstaben*** müssen aber ***nicht direkt aufeinander*** folgen, ist also nicht dasselbe wie ein Substring. Beispiel: ABCDEFGHIJK. Subsequenz: DFGHK, nicht Subsequenz: DAGH

10.3.1. Longest Common Subsequenz (LCS)

Gegeben sind zwei Strings X und Y. Finde die längste Subsequenz welche in X und in Y enthalten ist.
Beispiel: $X =$ ABCDEFG, $Y =$ ZACKDFWGH, LCS = ACDFG.
Anwendung: z.B. bei Vergleichen von DNA oder Source-Files mit Differenzen.

10.3.2. Brute-Force

Aufzählung aller Subsequenzen von X, testen, welche ebenfalls Subsequenzen von Y sind. Die längste Subsequenz als Resultat wählen. Sehr langsam, da $O(2^n)$.

10.3.3. LCS-Algorithmus

Definiere $L[i,j]$ als Länge der längsten gemeinsamen Subsequenz von $X[0..i]$ und $Y[0..j]$. Erlaube -1 als Index, sodass $L[-1,k]=0$ und $L[k,-1]=0$ um auszudrücken, dass der null Teil von X oder Y *(leeres Wort)* keine Übereinstimmung hat mit dem anderen. Definiere $L[i,j]$ folgendermassen:

- Wenn $x_i = y_i$ *(Übereinstimmung)*, dann $L[i,j] = L[i-1,j-1]+1$
- Wenn $x_i \neq y_j$ *(keine Übereinstimmung)*, dann $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$

			C	G	A	T	A	…	y_j
		-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		
…									
x_i	Index								

Beispiel:

j	0	1	2	3	4		i	0	1	2	3
Y	C	G	A	T	A	,	X	G	T	T	C

Bei $L[0,2]$ ist "G" $\neq$ "A", darum werden jetzt der Wert oberhalb ($L[-1,2]=0$) und links davon ($L[0,1]=1$) überprüft und der grössere davon (1) als neuer Wert eingetragen. Bei $L[1,3]$ ist "T" = "T", also wird der Wert diagonal links oberhalb mit 1 addiert ($L[0,2] = 1 \rightarrow 1+1=2$) und eingetragen.

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

```
Algorithm 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
```

		C	G	A	T	A	A	T	T	G	A	G	A
	L	-1	0	1	2	3	4	5	6	7	8	9	10	11
	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	1	1	1	1	1	1	1	1	1	1	1
T	1	0	0	1	1	2	2	2	2	2	2	2	2	2
T	2	0	0	1	1	2	2	2	3	3	3	3	3	3
C	3	0	1	1	1	2	2	2	3	3	3	3	3	3
C	4	0	1	1	1	2	2	2	3	3	3	3	3	3
T	5	0	1	1	1	2	2	2	3	4	4	4	4	4
A	6	0	1	1	2	2	3	3	3	4	4	5	5	5
A	7	0	1	1	2	2	3	4	4	4	4	5	5	6
T	8	0	1	1	2	3	3	4	5	5	5	5	6	6
A	9	0	1	1	2	3	4	4	5	5	5	6	6	6

Auslesen der Longest Common Subsequence

Die Lösung ist in $L[n,m]$ enthalten. Die gesuchte Subsequenz kann von der L-Tabelle ausgelesen werden. Beginne am Ende: $i=m-1, j=n-1$. Wenn das Zeichen für i und j gleich, füge $L[i,j]$ in LCS ein. Wechsle auf Position $L[i-1,j-1]$. Ansonsten: wie beim Erstellen der Tabelle $\max\{L[i-1,j], L[i,j-1]\}$. ***Rand um gleiche Zahlen einzeichnen, Ecken dieser Rahmen sind Pfadkanten.***

Beispiel:

i	0	1	2	3		j	0	1	2	3	4
X	G	T	T	C	,	Y	C	G	A	T	A

Wir starten mit $L[3,4]$. Die Zeichen matchen nicht, darum den Wert oberhalb ($L[2,4]=2$) und links davon ($L[3,3]=2$) überprüft. Da beide gleich gross sind, ist es egal, ob wir die Zeile oder Spalte wechseln. Hier gehen wir zu $L[3,3]$ und wiederholen das Ganze. Da $L[2,3]$ grösser ist, wechseln wir dahin. Bei $L[1,3]$ haben wir zum ersten Mal einen Match, packen darum "T" zuhinterst in LCS und gehen zu $L[0,2]$. Am Ende haben wir LCS = "GT".

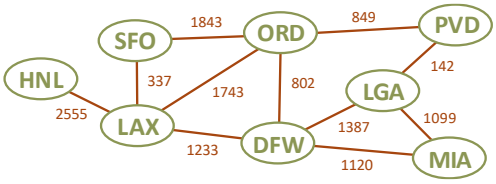
11. ZUSAMMENFASSUNG DYNAMISCHE PROGRAMMIERUNG

Algorithmus	*Zeitverhalten*	*Bemerkungen*
Knapsack: Brute-Force	$O(2^n)$	- Sehr schlecht weil exponentiell
Knapsack: Greedy		- Ergibt nicht immer optimale Lösung
Knapsack: Subprobleme	$O(n * m)$ n = *Anzahl Gegenstände,* m = *Anzahl Gewichte*	-
LCS: Brute-Force	$O(2^n)$	- Sehr schlecht weil exponentiell
LCS-Algorithmus	$O(n * m)$ *(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

12. GRAPHEN

Ein Graph ist ein Paar (V, E), wobei ***V*** ein Set von **Vertizes** (Knoten) ist und ***E*** 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.



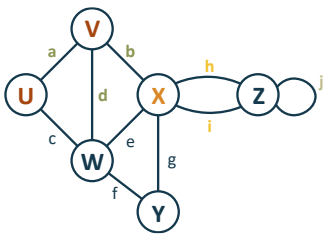
12.1. KANTEN-TYPEN

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

12.2. ANWENDUNGEN

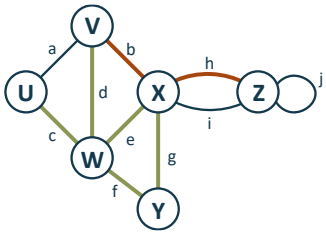
- ***Elektronische Schaltungen:*** Printed Circuit Board, Integrated Circuit
- ***Transport-Netzwerke:*** Autobahnnetz, Flugnetz
- ***Computer Netzwerke:*** LAN, Internet, Web
- ***Datenbanken:*** Entity-Relationship Diagramm

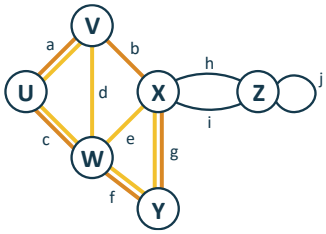
12.3. TERMINOLOGIE



- 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). $\mathbf{P_1 = (V, b, X, h, Z)}$
- ***Nicht einfacher Pfad:*** Ein Pfad, in dem ein Vertex oder eine Kante mehrfach passiert wird. $\mathbf{P_2 = (U, c, W, e, X, g, Y, f, W, d, V)}$
- ***Zyklus:*** Zirkuläre Sequenz - Start und Ziel sind gleich.
- ***Einfacher Zyklus:*** Ein Zyklus, sodass alle seine Vertizes und Kanten unterschiedlich sind. $\mathbf{P_3 = (V, b, X, g, Y, f, W, c, U, a)}$
- ***Nicht einfacher Zyklus:*** Ein Zyklus, in dem nicht alle Vertizes und Kanten unterschiedlich sind. $\mathbf{P_4 = (U, c, W, e, X, g, Y, f, W, d, V, a)}$





12.4. EIGENSCHAFTEN

Notation: $n =$ Anzahl Vertizes, $m =$ Anzahl Kanten, $\deg(v) =$ Grad von Vertex v

12.4.1. Eigenschaft 1

Der ***Grad eines Vertex*** ist gleich ***zweimal die Anzahl Kanten***. $\sum_v \deg(v) = 2m \Rightarrow n * \deg(v) = 2m$.

Beweis

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

Beispiel: $\boldsymbol{n = 4,\ m = 6,\ n * \deg(v) = 2m \Rightarrow 2m/n = \deg(v) \Rightarrow 12/4 = 3 \Rightarrow \deg(v) = 3}$

12.4.2. Eigenschaft 2

In einem ungerichteten, einfachen Graphen *(ohne Schleifen und ohne parallele Kanten)* gilt: $\boldsymbol{m \leq \big(n(n-1)\big)/2}$.

Bei einem voll vermaschten Graphen *(jeder Vertex ist mit jedem anderen Vertex verbunden)* ist $m = \big(n(n-1)\big)/2$.

Beweis

Jeder Vertex besitzt ein Grad von höchstens $(n-1)$.

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 $n-1$. Wie viele Kanten bleiben noch übrig vom nächsten Vertex zu den restlichen? Zu allen ausser sich selbst und zum ersten Vertex, also $n-2$. Daraus ergibt sich für einen voll vermaschten Graph mit 4 Vertizes:

$$(n-1) + (n-2) + (n-3) + (n-4) = \frac{n^2+n}{2} - n = \frac{n^2+n-2n}{2} = \frac{n^2-n}{2} = \frac{n(n-1)}{2}$$

12.4.3. 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.

12.4.4. Eigenschaft 4

Bei einem einfach verbundenen Graphen mit n ***Knoten und*** m ***Kanten ist*** $\mathrm{O(log(m)) = O(log(n))}$.

Beweis

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

- ***Untere Schranke*** $\underline{\mathrm{m}}$***:*** $n-1$ *(Jeder Vertex ist nur mit einem anderen verbunden - Liste)*
- ***Obere Schranke*** $\overline{\mathrm{m}}$***:*** $\frac{n(n-1)}{2}$ *(Voll vermaschter Graph)*

Somit muss bewiesen werden, dass $\log(\overline{m}) \in O(\log(n)) \cap \log(n) \in O(\log(\underline{m}))$

Beweis von $\log(\overline{m}) \in O(\log(n))$: Konstanten und niederwertige Terme streichen ($\frac{n(n-1)}{2} \Rightarrow \frac{n^2-n}{2} \Rightarrow n^2$)
$\log(n^2) \in O(\log(n)) \Rightarrow 2 * \log(n) \in O(\log(n)) \Rightarrow \log(n) \in O(\log(n))$

Beweis von $\log(n) \in O(\log(\underline{m}))$: Konstanten streichen ($n - 1 \Rightarrow n$)
$\log(n) \in O(\log(n))$

12.5. HAUPT-METHODEN DES GRAPH ADT (ABSTRACT DATA TYPE)

Vertizes (v, w) und Kanten (e) sind Positionen und speichern Elemente.

Zugriffs-Methoden

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

Update-Methoden

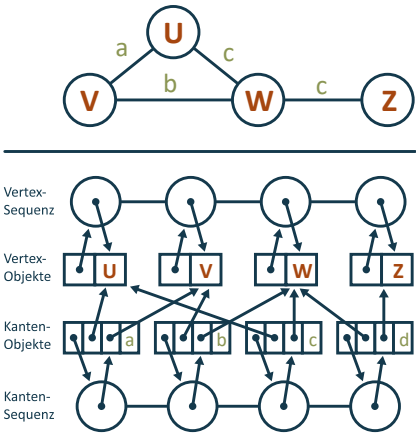
- `insertVertex(o):` Fügt einen Vertex mit dem Element o ein, gibt den neuen Vertex zurück
- `insertEdge(v,w,o):` Fügt eine neue Kante zwischen v und w ein, welches das Element o beinhaltet, gibt die neue Kante zurück
- `removeVertex(v):` Entfernt den Vertex v und seine angrenzenden Kanten
- `removeEdge(e):`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

Iterator-Methoden

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

12.6. 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)*

12.6.1. Remove-Operationen

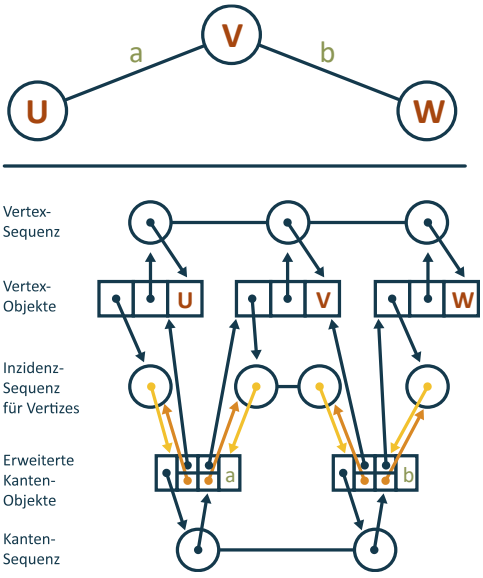
RemoveEdge()

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: $O(1)$

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 $O(n)$.

12.7. ADJAZENZ-LISTEN STRUKTUR



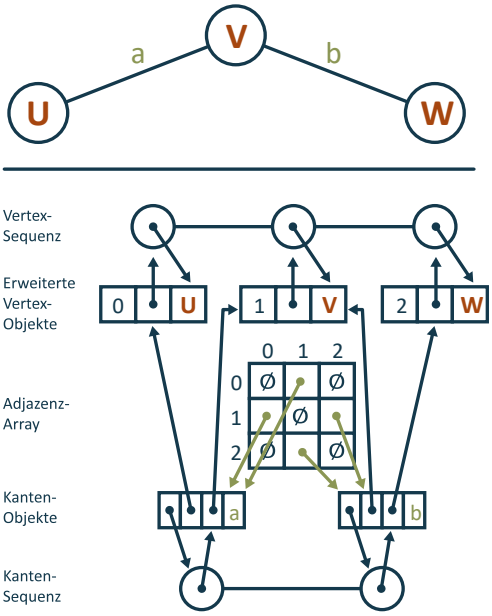
Erweiterung der Kanten-Listen Struktur um eine Inzidenz-Sequenz für jeden Vertex und erweiterte Kanten-Objekte.

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 $O(1)$ 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)*.

12.8. ADJAZENZ-MATRIX STRUKTUR



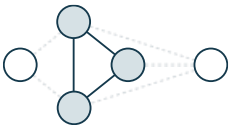
Erweitert die Kanten-Listen Struktur um einen ***2D-Array*** (Adjazenz-Array).

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

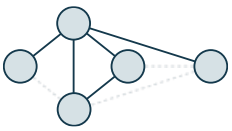
Der Adjazenz-Array referenziert auf die Kantenobjekte für adjazente Vertizes. `null` (Ø) für nichtadjazente Vertizes.

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

12.9. SUBGRAPHEN

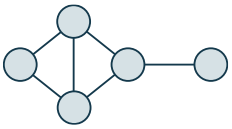


Ein ***Subgraph*** S eines Graphen G ist ein Graph, bei dem die Kanten von S eine Teilmenge der Kanten von G sind und die Vertizes von S eine Teilmenge der Vertizes von G sind.

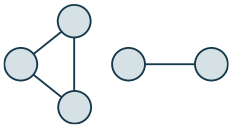


Ein ***spanning (aufspannender) Subgraph*** A des Graphen G ist ein Subgraph, welcher alle Vertizes, aber nicht alle Kanten von G enthält.

12.10. CONNECTIVITY

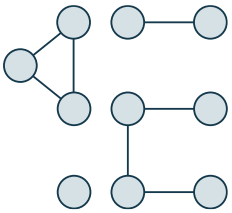


Ein Graph heisst ***verbunden*** (connected), falls zwischen ***jedem Paar von Vertizes*** ein ***Pfad*** existiert. Eine verbundene Komponente eines Graphen G ist ein verbundener Subgraph von G.



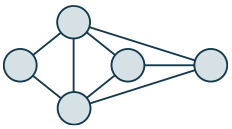
Es ist ***nicht möglich***, einen Graphen zu erstellen, der ***während dem Aufbau immer verbunden*** ist, da eine Kante erst erstellt werden kann, wenn beide dazugehörende Vertizes existieren.

12.11. BÄUME UND WÄLDER

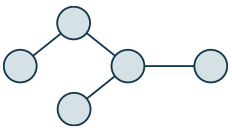


Ein (freier) ***Baum*** ist ein ungerichteter Graph T, sodass T ***verbunden*** ist und keine Zyklen aufweist. Anders als beim Wurzelbaum gibt es keinen Root-Knoten.
Ein ***Wald*** ist ein ungerichteter Graph ohne Zyklen. ***Die verbundenen Komponenten eines Waldes sind Bäume***. *Im Bild: 4 Bäume, die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

12.12. 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).*

13. ZUSAMMENFASSUNG PERFORMANCE GRAPHEN

n Vertizes, m Kanten, keine Parallelen Kanten, keine Schleifen	***Kanten Liste***	***Adjazenz-Liste***	***Adjazenz Matrix***
Platzverbrauch	$O(n+m)$	$O(n+m)$ *eigentlich* n + m + 2m	$O(n^2)$
`incidentEdges(v)`	$O(m)$	$O(\deg(v))$	$O(n)$
`areAdjacent(v,w)`	$O(m)$	$O(\min(deg(v), deg(w))$	$O(1)$
`insertVertex(o)`	$O(1)$	$O(1)$	$O(n^2)$
`insertEdge(v,w,o)`	$O(1)$	$O(1)$	$O(1)$
`removeVertex(v)`	$O(m)$	$O(\deg(v))$	$O(n^2)$
`removeEdge(e)`	$O(1)$	$O(1)$	$O(1)$

14. UNGERICHTETE GRAPHEN TRAVERSIERUNG

14.1. UNGERICHTETE TIEFENSUCHE / UNDIRECTED DEPTH-FIRST SEARCH

Eine DFS-Traversierung eines ungerichteten Graphen G

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

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.

14.1.1. Funktionsweise

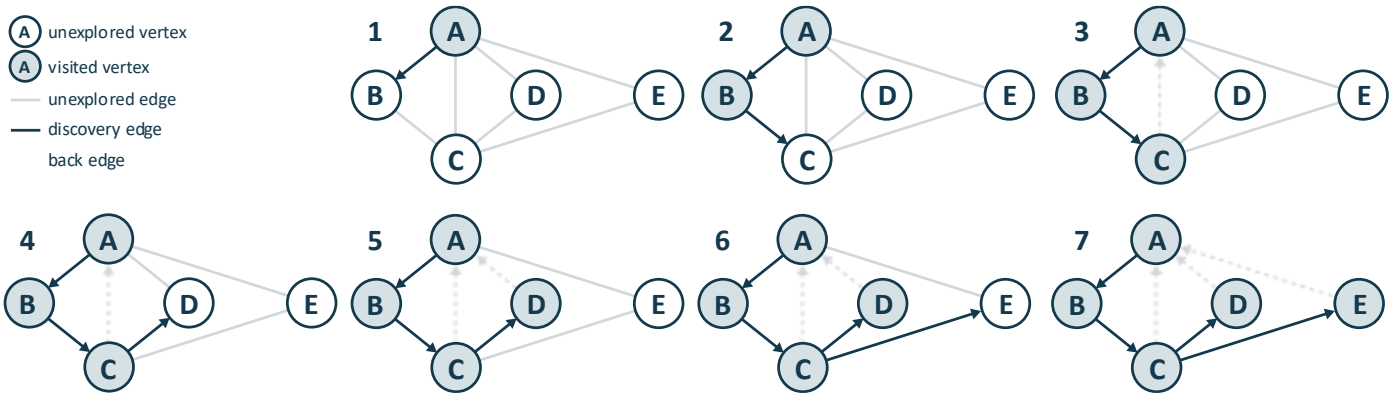
DFS vergibt allen Kanten und Vertizes ein Label. Zuerst werden alle Komponenten auf `UNEXPLORED` gesetzt. Gelangt DFS zu einem neuen Vertex, wird dieser als `VISITED` gekennzeichnet. Dann werden alle Kanten dieses Vertex mit `incidentEdges()` geholt und mit der «kleinsten» `UNEXPLORED` Kante begonnen.
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 labeling 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.



14.1.2. DFS und Labyrinth

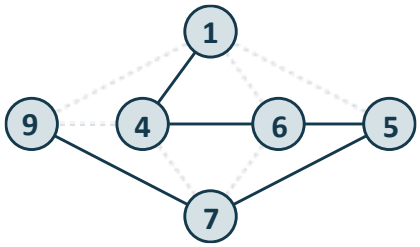
Der DFS-Algorithmus ähnelt der klassischen Strategie zur ***Erkundung eines Labyrinths***. Wir markieren jede ***besuchte Kreuzung***, ***Ecke*** und ***Sackgasse*** (Vertex), wir markieren jeden ***besuchten Korridor*** (Kante), und wir ***notieren den Rückweg*** zum Eingang (Start Vertex, Rekursion).

14.1.3. Eigenschaften von DFS

- `DFS(G,v)` besucht alle Vertizes und Kanten in der verbundenen Komponente von G beginnend bei v
- Die von `DFS(G,v)` markierten, besuchten Kanten bilden einen aufspannenden Baum für die verbundene Komponente von G beginnend bei v.

14.1.4. Pfade finden

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

14.1.5. Zyklen finden

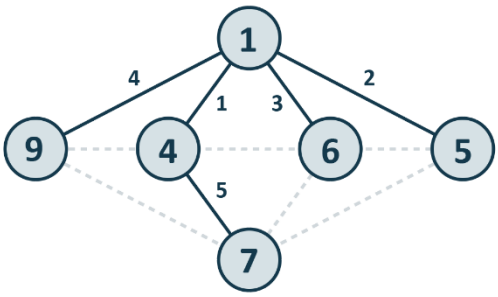
Der DFS kann auch darauf spezialisiert werden, ***einfache Zyklen zu finden***. Mithilfe eines ***Stacks*** S wird der ***Pfad*** zwischen dem Startvertex und dem aktuellen Vertex ***gespeichert***. Sobald ein ***Back-Edge(v,w)*** angetroffen wird, wird der ***Zyklus als Teil des Stacks ausgegeben***: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

```
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
        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()
```

14.2. BREITENSUCHE / BREADTH-FIRST SEARCH

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

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



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.

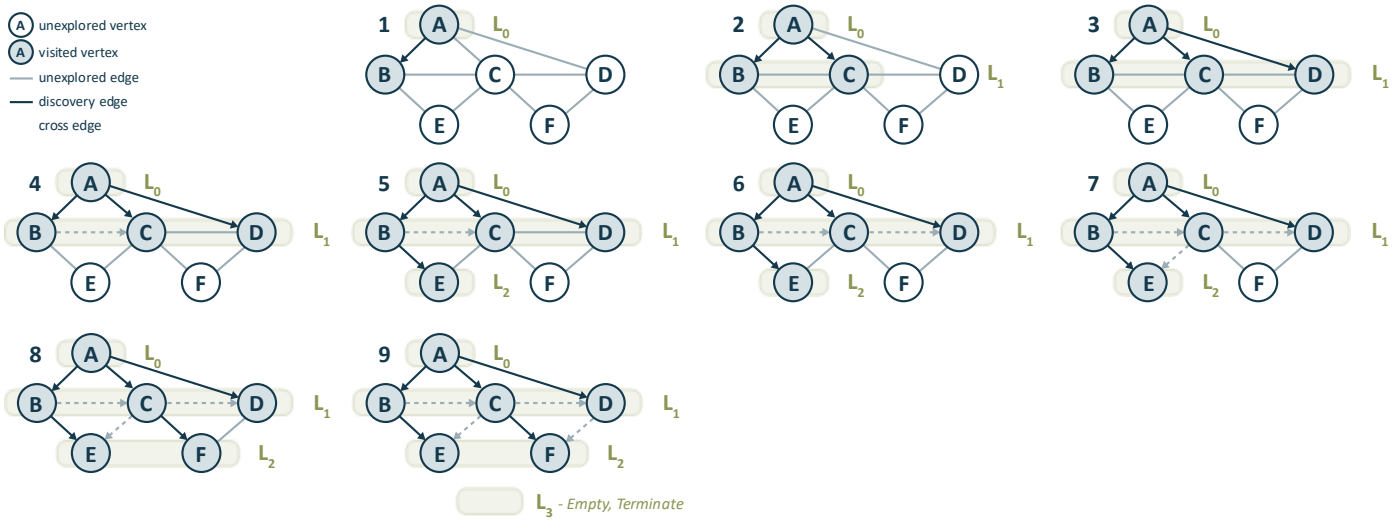
14.2.1. Funktionsweise

BFS geht ***der Reihe nach alle gegenüberliegenden Vertizes*** vom Startvertex durch und baut sich eine Liste von Listen L auf. In L_0 befindet sich nur der Startvertex. Nun werden ***alle Vertizes***, die mit dem ***Startvertex verbunden sind***, in L_1eingefügt. Der ***Index der Listen*** gibt an, ***wie viele Kanten*** dieser Vertex vom Startvertex ***entfernt*** ist (alle Vertizes in L_3 sind 3 Kanten vom Startvertex entfernt). In L_0 befindet sich nur der ***Startvertex***. Der Algorithmus ***terminiert***, nachdem alle Vertizes einer Liste L_i besucht wurden und es keine weiteren in L_{i+1} 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)
```

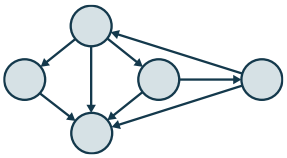
```
Algorithm BFS(G,s) // for all edges of a
vertice, *not* recursive
  L₀ ← new empty sequence
  L₀.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while !Lᵢ.isEmpty(): Lᵢ₊₁ ← new empty
sequence // run while there are verticies at
this level
    for all v ∈ Lᵢ.elements()
      for all e ∈ G.incidentEdges(v)
        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
```



14.2.2. Eigenschaften von BFS

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

15. 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)*.

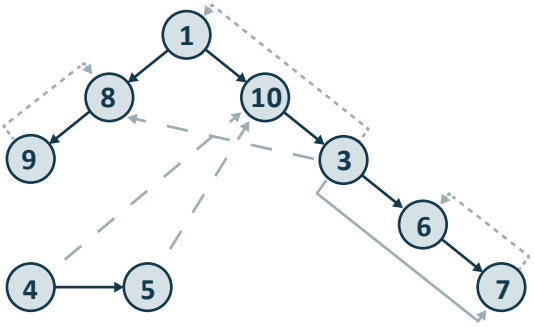
15.1. EIGENSCHAFTEN

- Ein Digraph ist: $G = (V, E)$ derart, dass jede ***Kante nur in eine Richtung*** geht. Kante (a, b) geht von a nach b, aber nicht von b nach a.
- Wenn G einfach *(maximal eine Kante von & zu demselben Vertex)* ist $\mathrm{m} \leq \mathrm{n}(\mathrm{n}-1)$ *(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

15.3. 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/Rückwertskante ist der Wert der Vertizes entscheidend ($A < B < C$ …). Deshalb hilft es, den Graphen als Baum zu zeichnen.

15.4. ERREICHBARKEIT / CONNECTIVITY

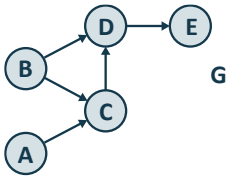
Je nachdem, welcher Vertex als Startpunkt gewählt wird, können eventuell nicht alle Vertizes traversiert werden. ***Strong Connectivity*** *(streng verbunden)* bedeutet, dass ***jeder Vertex alle anderen Vertizes erreichen*** kann.

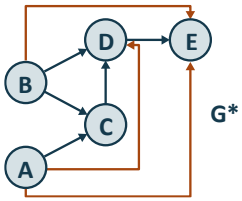
Überprüfungsalgorithmus: Ist Strong Connected?

- Wähle einen Vertex v in G. Führe eine ***Tiefensuche*** von v in G durch.
- Wenn es einen ***nicht besuchten Vertex*** w gibt: `return false`.
- G' sei G mit ***umgekehrten Kanten*** (Richtungen). Führe eine ***Tiefensuche*** von v in G' durch.
- Wenn es einen ***nicht besuchten Vertex*** w 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.

15.5. TRANSITIVER ABSCHLUSS / FLOYD-WARSHALL





Ist eine Verbindung transitiv, ist ein Vertex nur indirekt über mindestens einen anderen Vertex erreichbar. Ein Graph G wird mit ***direkten Pfaden ergänzt***, sodass alle erreichbaren Vertizes direkt erreichbar sind. Dies ergibt den Graphen G^*. 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 $a \to b \ \cap \ b \to c \Longrightarrow a \to c$

Floyd-Warshall

- Nummeriert die Vertizes von G als $v_1, \dots, v_n$ und berechnet eine Serie von Digraphen $G_0, \dots, G_n$, wobei $G_o = G$
- G_k hat eine gerichtete Kante (v_i, v_j), falls G einen gerichteten Pfad von v_i nach v_j mit Zwischenvertex aus der Menge $\{v_1, \dots, v_k\}$ hat.
- Es gilt: $G_n = G^*$.
- In der Phase k, Digraph G_k ist aus G_{k-1} berechnet.
- Laufzeit: $O(n^3)$ mit Adjazenz-Matrix.

Funktionsweise

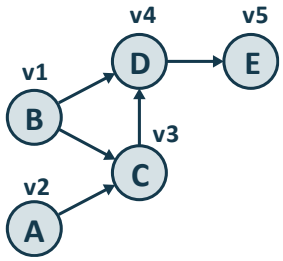
Alle Vertizes werden ***nummeriert***. Es sei $k = 1, i = 2, j = 3$. Zuerst wird geprüft ob die ***Kanten*** von $v_i \rightarrow v_k$ && $v_k \rightarrow v_j$ ***existieren***. Ist dies nicht der Fall, wird j bis n ***inkrementiert***, dann $i..n$ und zuletzt $k..n$. Wird eine solche ***transitive Verbindung*** gefunden, wird eine ***direkte Kante*** $v_i \rightarrow v_j$ ***eingefügt***, wenn noch nicht vorhanden.

Faustregel für Aufgaben: Gibt es Pfeile zu k und von k 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
```

15.6. 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 $v_1, \ldots, v_n$ der Vertizes, sodass für jede Kante (v_i, v_j) gilt: $\boldsymbol{i < j}$ (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
```

16. DFS VS. BFS

16.1. PERFORMANCE

Graphen mit n Vertizes und m Kanten

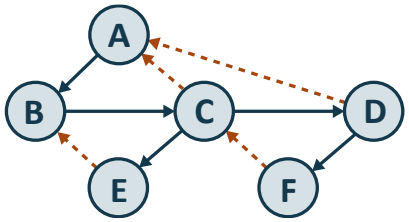
Algorithmus	*Zeitverhalten*
DFS	$O(n+m)$ *Setzen / Lesen eines Labels benötigt* $O(1)$ *Zeit* *Jeder Vertex wird zweimal markiert, zuerst als unexplored, dann als visited.* *Jede Kante wird zweimal markiert, zuerst als unexplored, dann als discovery oder back.* *Die Methode incidentEdges wird pro Vertex einmal aufgerufen. Mit Adjazenzlisten-Struktur.* *Laufzeit gilt auch für Strong Connectivity.*
BFS	$O(n+m)$ *Setzen / Lesen eines Labels benötigt* $O(1)$*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* L_i *eingetragen.* *Die Methode incidentEdges wird pro Vertex einmal aufgerufen. Mit Adjazenzlisten-Struktur.*
Floyd-Warshall	$O(n(n+m))$
Topological Sort	$O(n+m)$

16.2. APPLIKATION

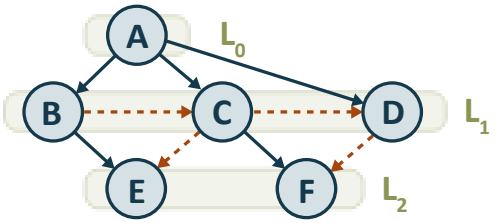
Applikationen	*DFS*	*BFS*
Aufspannender Wald, Verbundene Komponenten, Pfade, Zyklen	×	×
Kürzester Pfad		×
Erkennen von Biconnected Komponenten *(Kein Cut-Vertex: Graph kann nicht durch das Entfernen eines einzigen Vertex in zwei Teile zerfallen)*	×	

16.3. BACK EDGE VS CROSS EDGE

Back edge (v, w) *Rückwärtskante*
w ist ein Vorfahre von v im Baum der Suchkanten

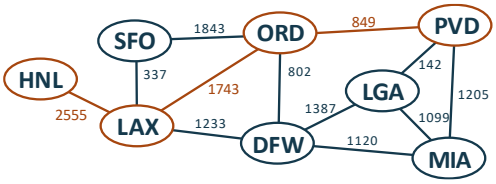


Cross edge (v, w) *Kreuzungskante*
w ist auf der selben Stufe wie v oder auf dem nächsten Level im Discovery-Kantenbaum



17. 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.



17.1. EIGENSCHAFTEN

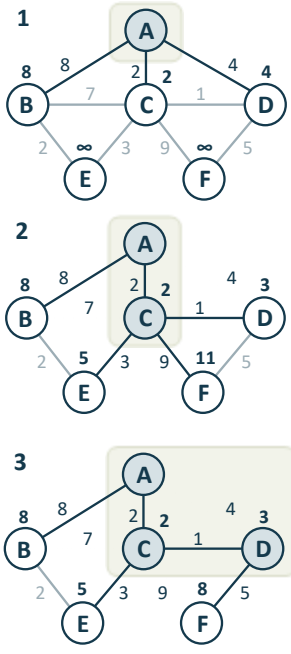
- Ein ***Teilweg*** eines kürzesten Weges ist selbst auch ein ***kürzester Weg***
- Es existiert ein ***Baum von kürzesten Wegen*** von einem Start-Vertex zu allen anderen Vertizes.

17.2. DIJKSTRA'S ALGORITHMUS

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

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

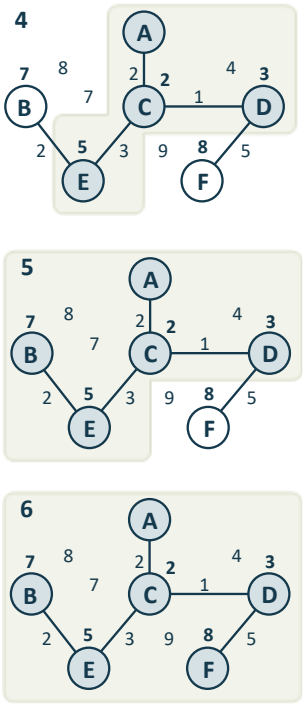
Kanten-Relaxation: Entspannung auf kürzeren Weg. Nachdem u der Wolke hinzugefügt wurde, wird bei den Nachbarskante e zum Vertex z die Distanz folgendermassen aktualisiert: $d(z) = \min(d(z),\ d(u) + weight(e))$. Die Baumkanten können sich noch ändern, solange noch nicht beide Vertizes einer Kante in 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.

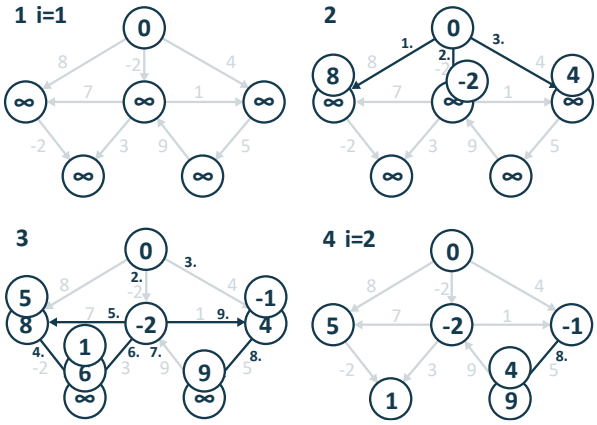
Alle Vertizes werden mit ihrer Distanz-Info in eine adaptable Priority Queue Q gepackt. In jedem Schritt wird der kleinste Vertex *(nach Distanz, dann nach Vertex-Reihenfolge)* aus Q entfernt und eine Relaxion durchgeführt. Damit ist dieser nun in der Wolke.




```
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)
```

17.3. 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 i-Kanten vom Startpunkt entfernt sind, minimal (=korrekt) sind.

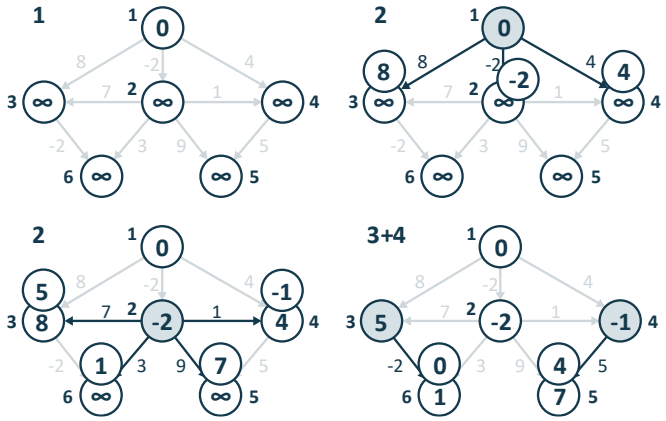



```
Algorithm BellmanFord(G,s)
 for all v ∈ G.vertices()
  if v = s: setDistance(v,0)
  else: setDistance(v, ∞)
 for i ← 1 to n-1 do
  for each e ∈ G.edges() //relax edge e
   u ← G.origin(e)
   z ← G.opposite(u,e)
   r ← getDistance(u) + weight(e)
   if r < getDistance(z)
    setDistance(z,r)
```

17.4. 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)
```



18. MINIMUM SPANNING TREES (MST)

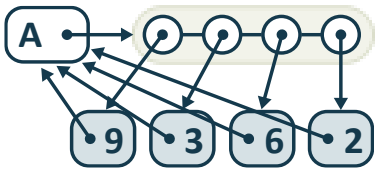
Ein minimum spanning tree ist ein aufspannender Baum eines Graphen mit ***minimalem totalen Kantengewicht***, der ***alle Vertizes beinhaltet*** *(Ein Graph, indem von einem Vertex jeder andere Vertex mit dem kleinstmöglichen Gewicht erreicht werden kann).* ***Anwendung:*** Kommunikationsnetzwerke, Transportnetzwerke.

18.1. EIGENSCHAFTEN

- ***Schlaufen-Eigenschaft:*** Gegeben: MST T eines Graphen G, Kante e, die nicht Teil des MSTs ist und Schlaufe C, die durch e mit T entsteht. Kann durch eine Kante e eine Schlaufe C entstehen, muss für jede Kante f in C überprüft werden, ob $weight(f) \leq weight(e)$. Falls nein, kann f durch e in T 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. e ist die Kante mit dem kleinsten Gewicht n, aber nicht im MST. Die Kante f ist grösser als e, aber momentan im MST. Kreiert man eine Schlaufe C zwischen den Teilmengen, e und f und wendet die Schlaufen-Eigenschaft an, wird f im MST durch e ersetzt, ohne dass sich die anderen Kanten im MST ändert.

18.2. 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.



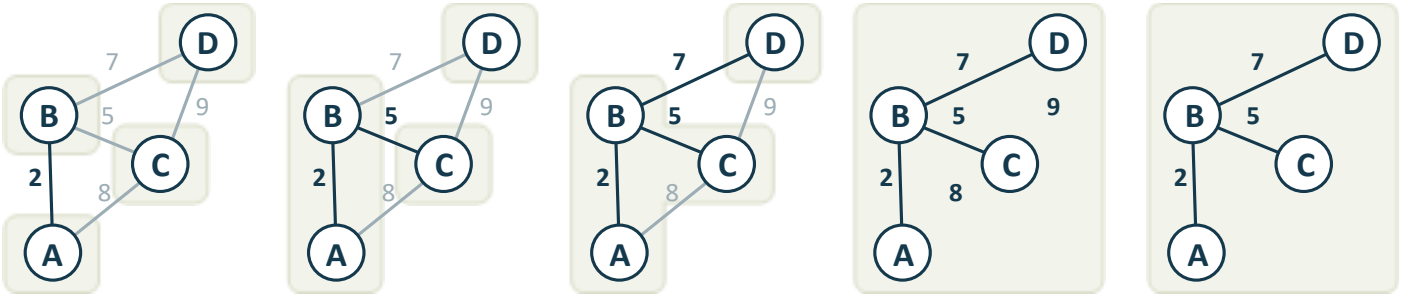
18.2.1. Repäsentation einer Partition

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 $O(1)$. 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 $O(\min(u.count(), v.count())$. Jedes verschobene Element

geht in ein Set mit mindestens ***doppelter Grösse*** als das bisherige, darum wird es höchstens $\log(n)$ mal verarbeitet.

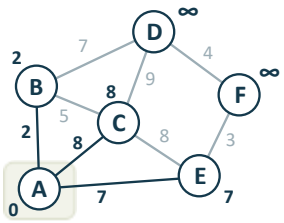
```
Algorithm KruskalMST(G)
 for every Vertex V in G do
  define a Cloud(v) of ← {v}
 Q //a Priority Queue. Add all the edges to Q
 T ← ∅
 while T < n-1 edges do
  edge e = Q.removeMin()
  u, v: Endpoints of e
  if Cloud(v) != Cloud(u) then
   add Edge e to T
   merge Cloud(v) and Cloud(u)
 return T
```

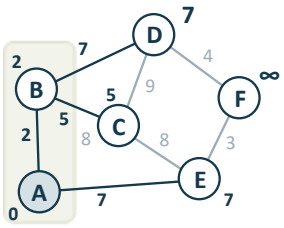
```
Algorithm Kruskal ( G)
 Input Ein gewichteter Graph G.
 Output Ein MST T für G.
 // P sei eine Partition der Vertizes von G, wobei
jeder Vertex ein Set für sich bildet
 // Q sei eine Priority Queue, welche die Kanten
von G nach Gewichtung sortiert
 // T sei ein ursprünglich leerer Baum
 while Q is 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
```

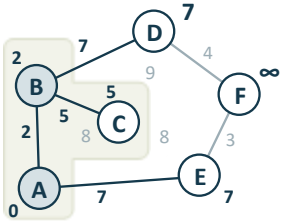


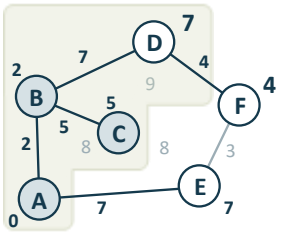
18.3. PRIM-JARNIK'S ALGORITHMUS

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.









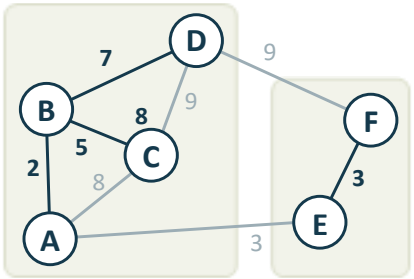

```
Algorithm PrimJarnikMST(G)
 Q ← new heap-based adaptable PQ
 Cloud ← new Hash-Set
 s ← a vertex of G // irgendein Vertex als Startvertex
 for all v ∈ G.vertices()
  if v = s: setDistance(v,0)
  else: setDistance(v, ∞)
  setParent(v, ∅)
  l ← Q.insert(getDistance(v),v)
  setLocator(v,l)
 while !Q.isEmpty()
  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)
```

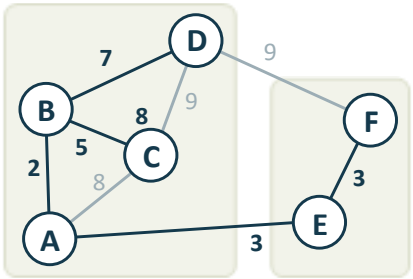
18.4. BORUVKA'S ALGORITHMUS

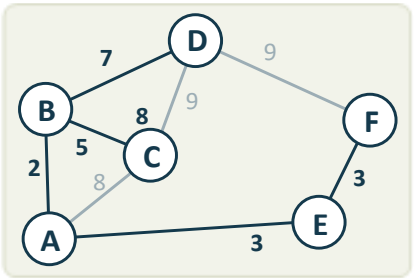
Bildet wie der Kruskal's Algorithmus viele Wolken aufs Mal. Jede Iteration der while-Schlaufe halbiert die Anzahl der verbundenen Komponenten in T. Pro Wolke gibt es eine Priority Queue.

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

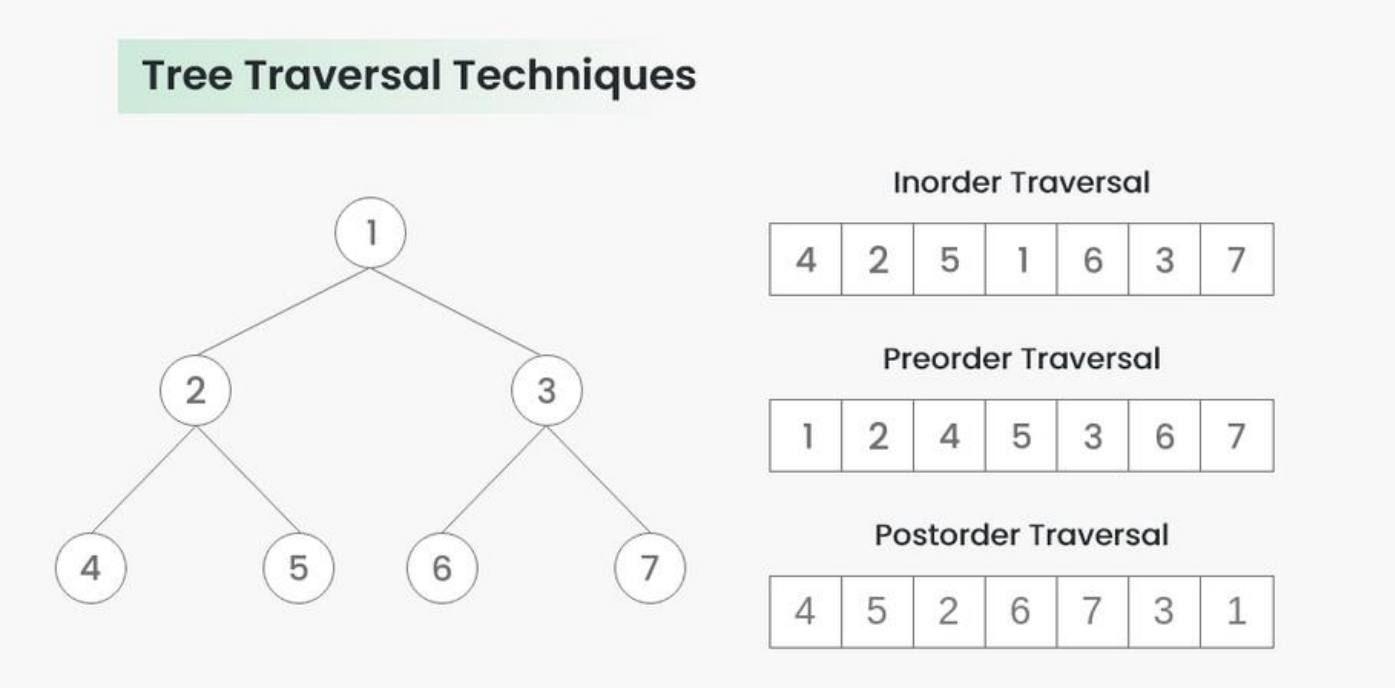
```
Algorithm BoruvkaMST(G)
 T ← V {nur die Vertizes von G}
 while T < n-1 edges do
  for each verbundene Komponente C in T do
    if e is not in T then //Edge e is the smallest Edge in C to another component in T
     add e to T
  return T
```







19. TRAVERSIERUNGSARTEN



20. O-NOTATIONSLISTE

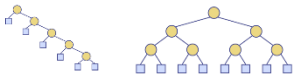
Multimap / Suchtabelle

find(k)	insert(k)	remove(k)
$O(\log n)$	$O(n)$	$O(n)$

Binary Search Tree

Speicherplatz	find(k)	insert(k)	remove(k)
$O(n)$	$O(h)$	$O(h)$	$O(h)$

Die Höhe h ist $O(n)$ im schlechtesten Fall (Entarteter Baum) und $O(log\ n)$ im besten Fall (Balanced Tree)



AVL Tree

Restrukturierung	find(k)	insert(k)	delete(k)
$O(1)$	$O(\log n)$	$O(\log n)$ *wegen find()* *eventuelle Restrukturierung baumaufwärts ist* $O(1)$	$O(\log n)$ *wegen find()* *eventuelle Restrukturierung baumaufwärts sind* $O(log\ n)$

Splay Tree

$O(h)$

Durchschnitt: $O(\log n)$*, für oft besuchte Knoten schneller, da diese immer näher an die Root rücken. Worst-Case: Höhe des Baumes ist* n*, somit* $O(n)$ *Rotationen, jede mit* $O(1)$

Merge Sort

Höhe des Trees	Gesamt-Aufwand aller Knoten einer Tiefe i	Totale Laufzeit
$O(\log n)$	$O(n)$ *Aufteilung und Mischen von 2^i Sequenzen der Grösse $n/2^i$, 2^{i+1} rekursive Aufrufe*	$O(n * log\ n)$

Quick Sort

Partitio-nierung	Höhe im Optimalfall	Erwartete Laufzeit (durchschnittlich)	Worst-Case Laufzeit
$O(n)$	$O(\log n)$ *weil Halbierung auf jeder Stufe*	$O(n * \log n)$ *Tiefe im Optimalfall: $O(log\ n)$, Gesamtaufwand für alle Knoten einer Tiefe: $O(n)$*	$O(n^2)$ *wenn das Pivot das Minimum oder Maximum-Element ist. L oder G hat dann die Länge $n-1$, das andere 0. Die Laufzeit ist proportional zur Summe. $\sum_{i=0}^{n} i = \frac{n^2+n}{2}$* *Anzahl Vergleiche: $\frac{(n-1)n}{2}$*

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

Bucket Sort

Laufzeit einzelner Phasen	Erwartete Laufzeit	Worst-Case Laufzeit
$O(n)$ *Phase 1: Buckets füllen* $O(N+n)$ *Phase 2: Elemente aus allen Buckets herausnehmen*	$O(N+n)$ *n: Anzahl Elemente, N: Anzahl Buckets*	$O(n^2)$ *Wenn alle Elemente im selben Bucket landen*

Brute Force

$O(n * m)$

n:Textlänge, m: Patternlänge

Boyer Moore

Laufzeit	Last Occurrence Funktion
$O(n * (m+s))$ *n-mal die Last-Occurence-Funktion*	$O(m+s)$ m ist die Länge von P und s die Anzahl Zeichen im Alphabet Σ

KMP

Laufzeit	Failure Function
$O(m+n)$ *Bei jeder Iteration der while-schleife wird entweder* i *um eines erhöht oder die Verschiebung* $\mathrm{i}-\mathrm{j}$ *nimmt um mindestens* 1 *zu. Somit ergeben sich maximal* $2\mathrm{n}$ *iterationen in der while-Schleife.*	$O(m)$ *m ist die Länge von P* *Bei jeder Iteration der while-schleife wird entweder i um eines erhöht oder die Verschiebung $i-j$ nimmt um mindestens 1 zu. Somit ergeben sich maximal $2m$ Iterationen in der while-Schleife.*

Standard-Trie

Speicherplatz	find(k)	insert(k)	remove(k)
$O(n)$	$O(dm)$	$O(dm)$	$O(dm)$

n = totale Länge der Strings in S, m = Länge des String-Parameters der Operation, d = Grösse des Alphabets

Trie kompakte Repräsentation (Array)

$O(s)$ Speicherplatz

s = Anzahl Strings im Array

Suffix-Trie

Speicherplatz	Pattern Matching	Erstellen
$O(n)$	$O(d * m)$	$O(n)$

n = totale Länge des Strings S, m = Länge des Patterns, d = Grösse des Alphabets

Dijkstra

Laufzeit	Setzen / Lesen Labels	Einfügen / Löschen Priority Queue	Schlüssel ändern
$O\big((n+m) * \log n\big)$	$O(dez(z) * O(1)$ *Anzahl * Zeit*	$O(\log n)$	$\deg(w) * O(\log n)$

Bellman-Ford

$O(n * m)$

DAG-basierter Algorithmus

$O(n + m)$

Kruskal

Laufzeit	find	Union(u,v)	Max Anzahl Verarbeitungen pro Element
$O(m * \log n)$ *Partitionsbasiert*	$O(1)$	$\min(n_u, n_V)$ n_u *und* n_v *sind die Grössen der Sets, die* u *und* v *beinhalten*	$\log n$

Prim Jarnik

Laufzeit	Setzen / Lesen Labels	Einfügen / Löschen Priority Queue	Schlüssel ändern
$O\big((n+m) * \log n\big)$ *Mit Adjazenz-Listen Struktur*	$O(dez(z) * O(1)$ *Anzahl * Zeit*	$O(\log n)$	$\deg(w) * O(\log n)$

Das ABC für die Besitzer eines $O(n^2)$-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