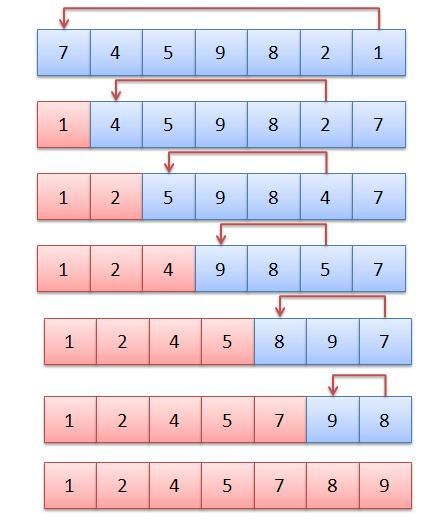
**1. Sortieren**

Um internes Sortieren handelt es sich, wenn sich die zu sortierenden Daten im Hauptspeicher befinden. 🡪 direkter schneller Zugriff

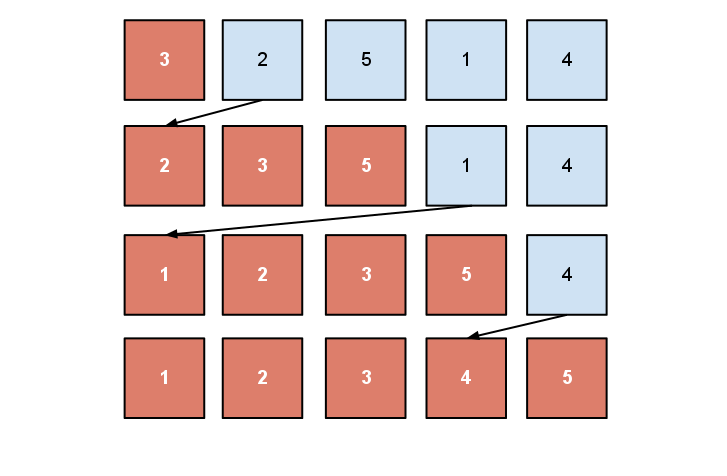
Von externem Sortieren spricht man, wenn die zu sortierenden Daten auf externen  
Speichermedien wie Bänder oder Platten ausgelagert sind. 🡪 Datenzugriff nur sequentiell (Bänder) oder in Blöcken (Festplatten)

**Selection Sort**

- Zwei verschachtelte Vorschleifen

- O(n2)

- Laufzeit abhängig vom Vorsortieren

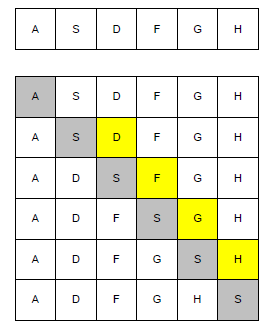
**Insertion Sort**

- Zwei verschachtelte Vorschleifen

- O(n2)

- Laufzeit abhängig vom Vorsortieren

**Bubble Sort**



- Durchläuft wiederholt die Daten und vertauscht falls

notwendig die Nachbarn

- Wenn Durchlauf ohne Tausch gelingt 🡪 Daten sortiert

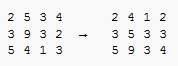
**Shell Sort**

Shellsort bedient sich prinzipiell des Insertionsorts (Optimierte Version). Es versucht den Nachteil auszugleichen, dass hier Elemente in der Sequenz oft über weite Strecken verschoben werden müssen. Dies macht Insertionsort ineffizient. Shellsort verfolgt den Ansatz, dass die Sequenz zuerst in einzelne Untersequenzen zerlegt wird und diese sortiert werden.

***2^n,...,4,2,1Beispiel****:*

Zu sortieren sind die Zahlen 2, 5, 3, 4, 3, 9, 3, 2, 5, 4, 1, 3 mittels der Folge

Zuerst werden die Daten in eine Matrix mit vier Spalten eingetragen und spaltenweise sortiert. Die Zahlenfolge wird also 4-sortiert.



Die sortierte Vier-Spalten-Matrix wird nun in zwei Spalten aufgeteilt, wobei von links nach rechts gelesen wird. Diese Spalten werden nun 2-sortiert.



Die sortierte Zwei-Spalten-Matrix wird nun in eine Spalte geschrieben und wieder sortiert mittels normalem Insertionsort. Der Vorteil dabei besteht darin, dass kein Element der Sequenz so weit verschoben werden muss, wie beim Insertionsort, der auf eine nicht vorsortierte Folge angewendet wird.



Die hier verwendete Schrittfolge 1,2,4,8,16,...,2^n(wie es 1959 original von Shell vorgeschlagen wurde) erweist sich in der Praxis allerdings als nicht zweckmäßig, da nur gerade Stellen sortiert werden und ungerade Stellen der Sequenz nur im letzten Schritt angefasst werden. Als zweckmäßiger hat sich 1,4,13,40, ... erwiesen.

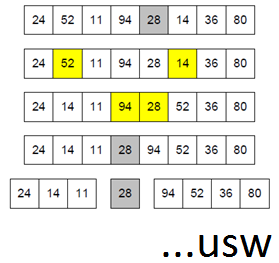
**Quick Sort**

Prinzip: teile die Daten in zwei Teile und sortiere diese unabhängig voneinander.

Zunächst wird die zu sortierende Liste in zwei Teillisten („linke“ und „rechte“ Teilliste) getrennt. Dazu wählt Quicksort ein sogenanntes Pivotelement aus der Liste aus. Alle Elemente, die kleiner als das Pivotelement sind, kommen in die linke Teilliste, und alle, die größer sind, in die rechte Teilliste. Die Elemente, die gleich dem Pivotelement sind, können sich beliebig auf die Teillisten verteilen. Nach der Aufteilung sind die Elemente der linken Liste kleiner oder gleich den Elementen der rechten Liste.

Anschließend muss man also nur noch jede Teilliste in sich sortieren, um die Sortierung zu vollenden. Dazu wird der Quicksort-Algorithmus jeweils auf der linken und auf der rechten Teilliste ausgeführt. Jede Teilliste wird dann wieder in zwei Teillisten aufgeteilt und auf diese jeweils wieder der Quicksort-Algorithmus angewandt, und so fort. Diese Selbstaufrufe werden als Rekursion bezeichnet. Wenn eine Teilliste der Länge eins oder null auftritt, so ist diese bereits sortiert und es erfolgt der Abbruch der Rekursion.

***Beispiel:***

- Mittlerer Aufwand O(N log N)

- Worst Case Aufwand O(N2)

- Worst Case verhalten kann durch geschickte Auswahl des Pivots elementpraktisch ausgeschlossen werden

**Heap Sort**

Ziel ist es eine Datenmenge so zu sortieren das immer das größte Element zugreifbar ist.

Heap sortiert Elemente durch Versickern.

Vergleiche beide Kind Elemente

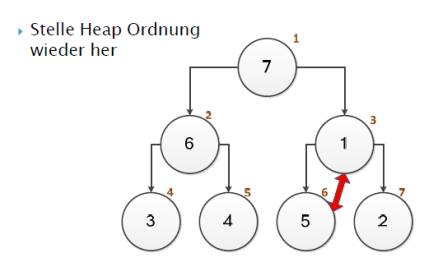
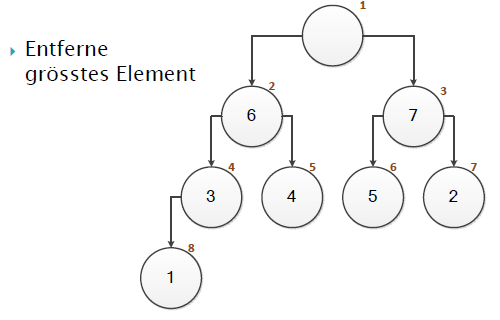
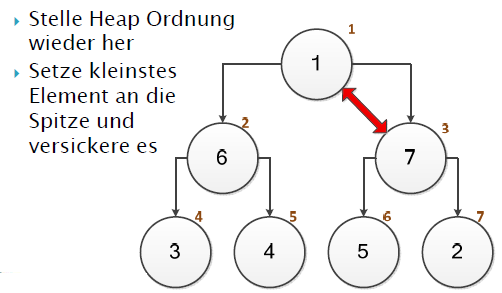
* Tausche mit größerem Platz
* Solange bis beide Elemente kleiner sind oder das ursprüngliche Element am Boden

angelangt ist.

***Beispiel sortieren:***

***Array: 3, 5, 1, 6, 7,2 ,0***

***Beispiel entferne größtes Element:***

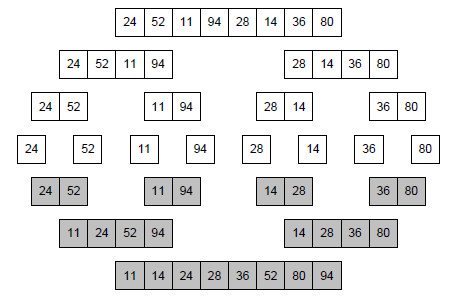
******

***Merge Sort***

Prinzip:

Daten werden in zwei Teile geteilt, jeder Teil wird durch rekursive Anwendungen sortiert. Einzeln sortierte Teile werden dann zusammengesetzt (merged).

***Beispiel:***

******

**2. Suchen**

***Lineare Suche***

- Daten werden sequentiell mit dem Schlüssel verglichen.

- Worst Case Aufwand: O(N)

- Vorsortierung nicht notwendig

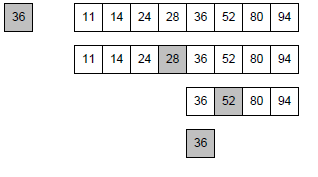
***Binärsuche***

- Nur bei sortierten Daten (nur bei Array anwendbar da Daten in direktem Zugriff stehen müssen)

- Daten werden schrittweise um die Hälfte eingeschränkt. (Ca. bei der hälfte des Arrays teilen)

- Aufwand O(log N)

***Beispiel:***

- Gesuchte Zahl 36

- Daten werden immer um die hälfte eingeschränkt usw.

***Fibonacci Suche***

Bei der Fibonacci Suche wird das Suchintervall mit Hilfe der Fibonaccizahl, ähnlich wie bei der binären Suche, schrittweise verkleinert. Aufwand: O(1,618n)

Fibonacci Reiheinfolge: { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …}

***Beispiel:***

Gesuchter Wert: **9**

Array: **{0, 8, 9, 50, 51, 52, 53, 55, 56, 57}**

Fibonacci Reiheinfolge: **{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …}**

Fibonacci: **8**

Obere Grenze: **9**

Untere Grenze: **0**

**Erklärung:**

Array Elemente: 10

Obere Grenze: **9** -> 10 – 1 Daher in der Fibonacci mit dem höchstmöglichen beginnen das ist in diesen Fall 8.

--------------------------------------------------------------------------------------------------------------------------------------

1. Array[Fibonacci] = 8

Die Zahl 8 ist **kleiner** als der gesuchte Wert d.h. wir müssen die Fibonacci Zahl auf die nächst kleinere reduzieren. Im Anschluss wird die Obere Grenze auf 8 festgelegt.

Fibonacci: **5**

Obere Grenze: **8**

Untere Genze: **0**

1. Array[Fibonacci] = 51

Die Zahl 51 ist **größer** als die gesuchte Zahl 9 d.h. die gesuchte Zahl muss sich also im Array index Bereich von 5 und 8 befinden. Falls die Obere Grenze Größer als die Aktuelle „Fibonacci Zahl + 1“ ist können wir die Untere Grenze auf „Fibonacci Zahl + 1“ festlegen sonst wird die Untere Grenze auf nur auf Fibonacci festgelegt.

Fibonacci: **7**

Obere Grenze: **8**

Untere Genze: **5**

1. Array[Fibonacci] = 9

Die Zahl 9 wurde gefunden und wir sind somit fertig.

***Exponentielle Suche***

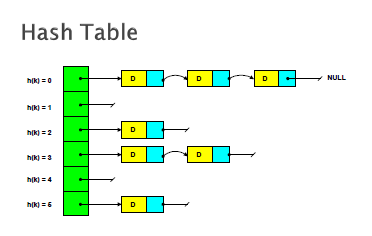
Für Fälle wo der Suchbereich unbekannt oder praktisch unbegrenzt. Es wird eine obere Grenze für den durchsuchenden Bereich bestimmt. Weitere Suche erfolgt dann äquivalent zu der Binär suche.

**2. Hashing**

**Motivation:** Sortiere Elemente in einem Array. Statt Suche werden mathematische Funktionen zum ermitteln der Slots verwendet.

**Alternativen zu Arrays:** **Hash Tables**

Statt Index wird ein Schlüssel verwendet, der duch eine hash Funktion berechnet wird. Eine Hash Funktion kann für mehrere Einträge den gleichen Wert liefern.

***Beispiel:***

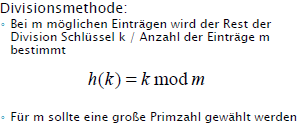
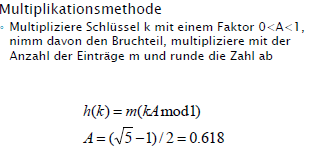
- Es gibt bestimmte Anzahl v von Einträgen (buckets)

- Beim Einfügen wird ein Key berechnet. Anschließend wird das Element in die Entsprechende Liste gehängt

***Hashing:***

Ziel des Hashings ist es eine möglichst gute Gleichverteilung in der Hash Table zu erzielen. Dafür müssen ungefähr die Anzahl der Elemente bekannt sein. Wenn zu viele Einträge muss die Hash Table neu berechnet werden.

Zwei Methoden: **Divisionsmethode** und **Multiplikationsmethode**



**Arten von Hashing:**

* Universelles Hasing 🡪 Solange Schlüssel kleiner ist als Anzahl der Speicherplätze kommt es zu

keinen Kollisionen.

Wenn doch müssen Überläufe in einer Liste gespeichert werden

* Offenes Hashing 🡪 Bei Kollisionen wird ein Alternativer Slot im Hash-Array bestimmt

Offenes Hashing sollte Benutzt werden wenn hauptsächlich eingefügt und gesucht, nicht aber gelöscht wird.

* Dynamisches Hashing 🡪 Für stark wachsende oder schrumpfende Datenbestände.

**2. Bäume**

Ein Baum ist ein Spezialfall eines Graphen. Er besteht aus

* Knoten (nodes)
* Blätter (leaves)
* Wurzel (root)

Alle Knoten eines Baumes sind genau durch einen Pfad miteinander verbunden. Die Höhe eines Baumes wird durch die Ebene der knoten festgelegt. Falls jeder Knoten eine bestimmte Anzahl an Nachfolgern hat spricht man von n-ären Bäumen:

* Binärer Baum (n = 2)
* Ternärer Baum (n = 3)

**Eigenschaften von Bäumen:** Für zwei beliebige Knoten in einem Baum existiert genau ein Pfad der sie

verbindet.

Ein Baum mit N Knoten hat N-1 Kanten.

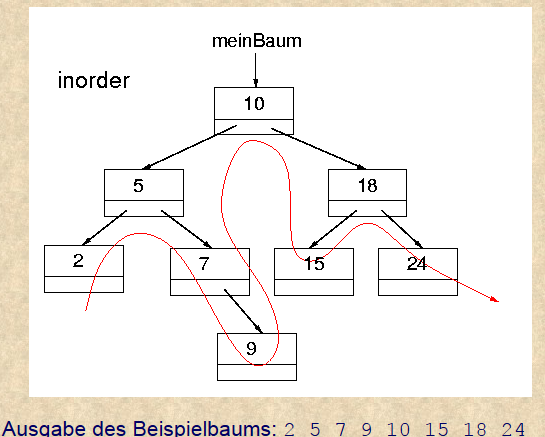
**Binäre Bäume:**

* Jeder Knoten hat genau einen Nachfolger
* Das Auffinden eines Elements erfordert O(logN) Zugriffe. Im entarteten Baum kann das bis zu O(N) ansteigen (Liste)

**Durchlaufen eines Binären Baumes:**

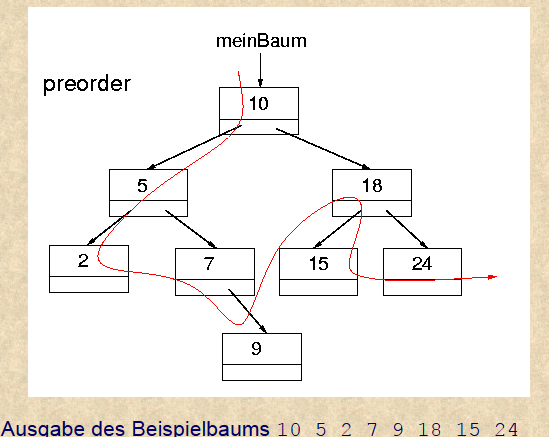
* **In Order**

Linker Ast, Knoten, rechter Ast



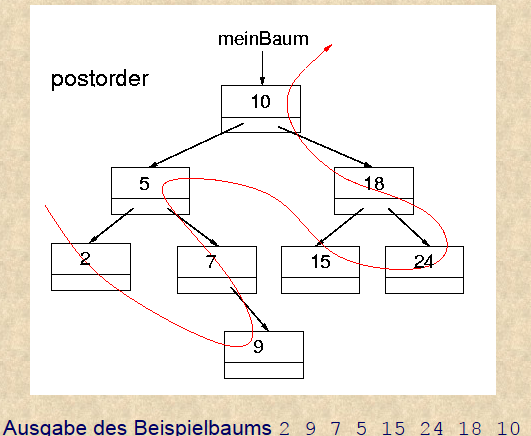
* **Pre Order**

Knoten, linker Ast, rechter Ast



* **Post Order**

Linker Ast, Rechter Ast, Knoten



**AVL Baum**

Probleme die bei Binären Baumen auftreten ist das sie nach löschen oder einfügen unausgeglichen werden, sprich nicht balanciert. AVL Baum ist ein Baum der einen weiteren Faktor berücksichtigt, nämlich den balance factor (bf). **bf = Höhe links – Höhe rechts**

* Bf > 0 🡪linker Ast ist länger
* Bf < 0 🡪 rechter Ast ist länger

Nach dem Einfügen oder Löschen müssen Elemente rotiert werden um eine möglichst gute Balance zu halten. Es gibt vier Mögliche Rotationen LL, LR, RL, RR Rotation.

**B- Baum**

***Beispiel:*** Einfügen 2,4,6,8,10,12

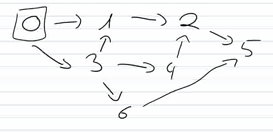
1) 2) 3)

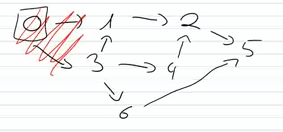
4) 5)

6)

**2. Graphen**

**Topologische Sortierung**

1. Wähle alle Knoten ohne Vorgänger 🡪0



1. Nimm die Kanten die von 0 weggehen

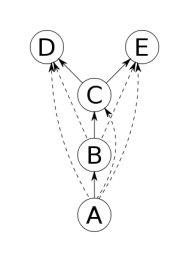
heraus und wiederhole Schritt eins

…

(Falls es in einem Graphen mehrere Knoten ohne Vorgänger gibt, wähle einen aus mit dem Begonnen wird bzw. mit welchem fortgesetzt werden kann)

**Topologische Reihenfolge: 0, 3, 1, 6, 4, 2, 5** (Es gibt mehrere Möglichkeiten)

**Transitive Hülle**

Die Transitive Hülle eines Graphen umfasst alle Knoten und Kanten, die wechselweise miteinander in Verbindung stehen. Alle von einem Punkt aus erreichbaren Knoten liegen in dieser Hülle.

***Beispiel:*** Gegeben sei eine Relation „Direkter-Vorgesetzter“ mit folgenden Beziehungen:

* C ist direkter Vorgesetzter von D und E
* B ist direkter Vorgesetzter von C
* A ist direkter Vorgesetzter von B

Die transitive Hülle dieser Relation enthält nun zusätzlich auch die indirekten Vorgesetzten:

* A ist Vorgesetzter von B, C, D, E
* B ist Vorgesetzter von C, D, E
* C ist Vorgesetzter von D und E

**Breitensuche:**

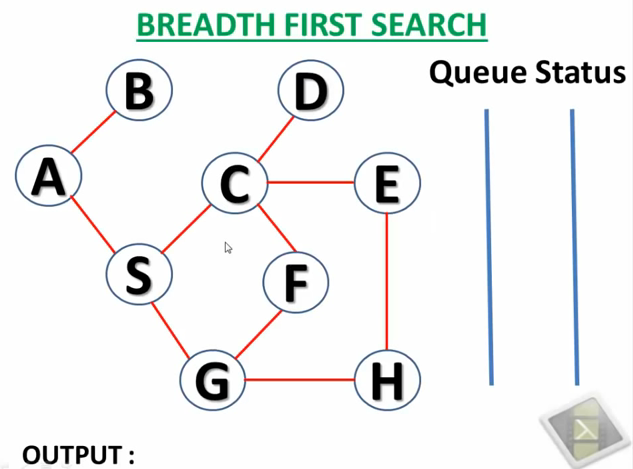
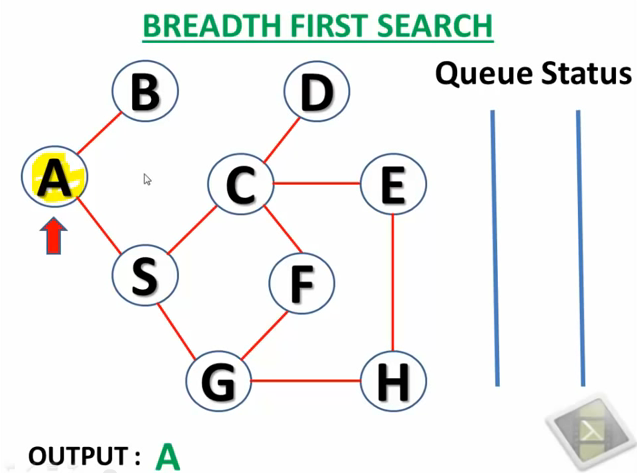
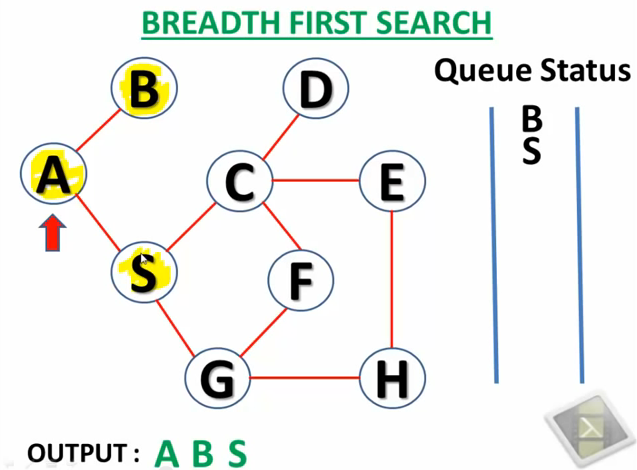
Wird zum durchlaufen des Graphen benötigt.

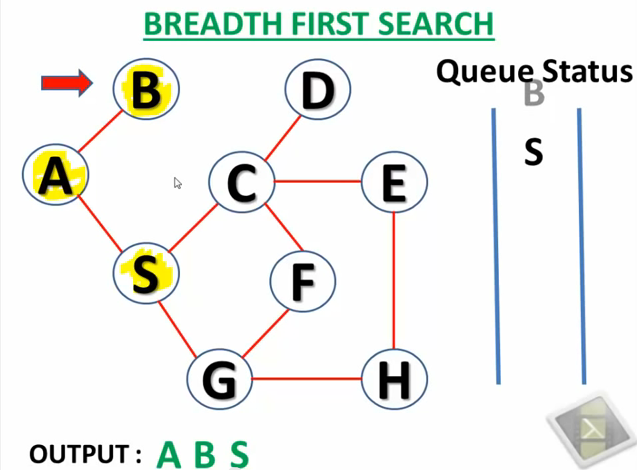
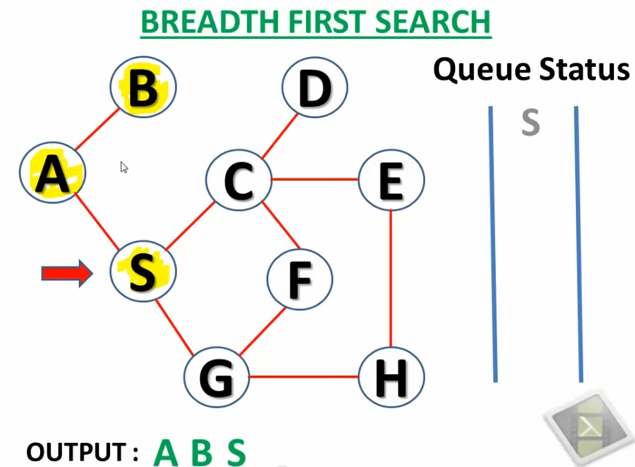
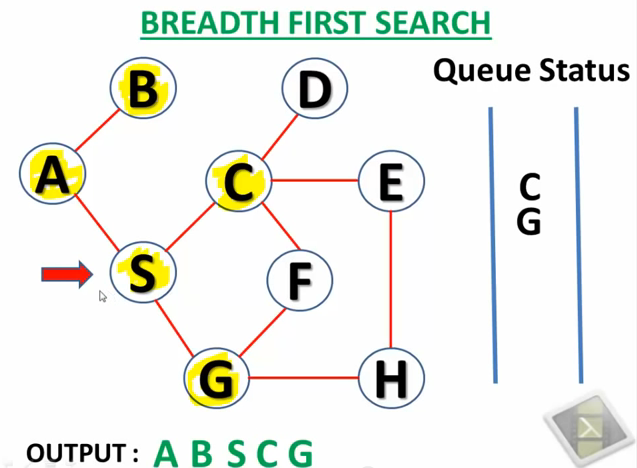
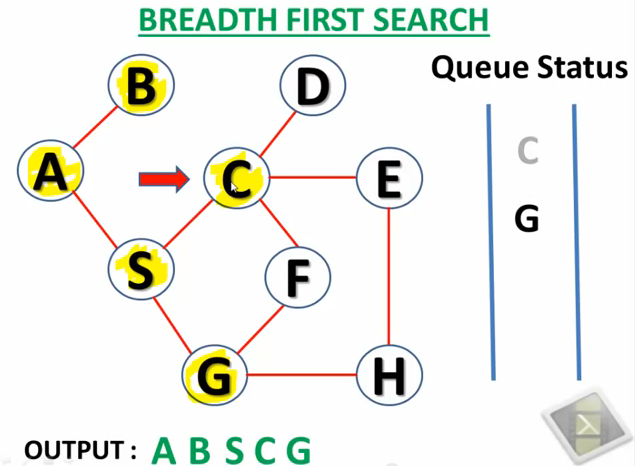
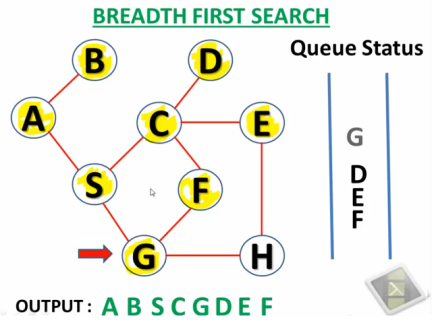
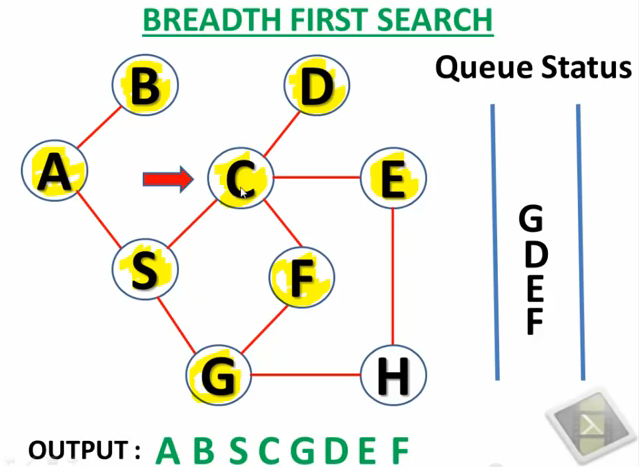
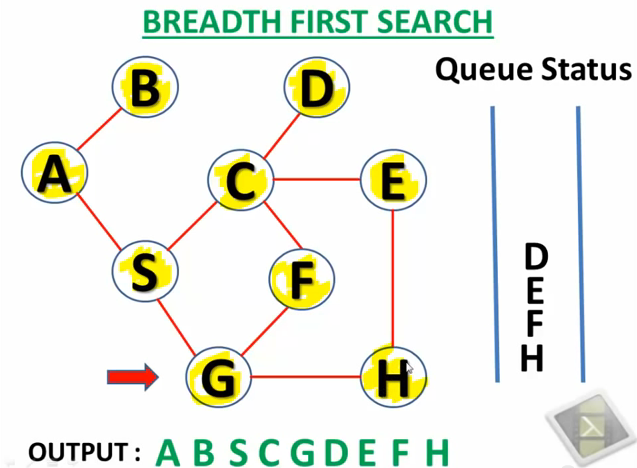
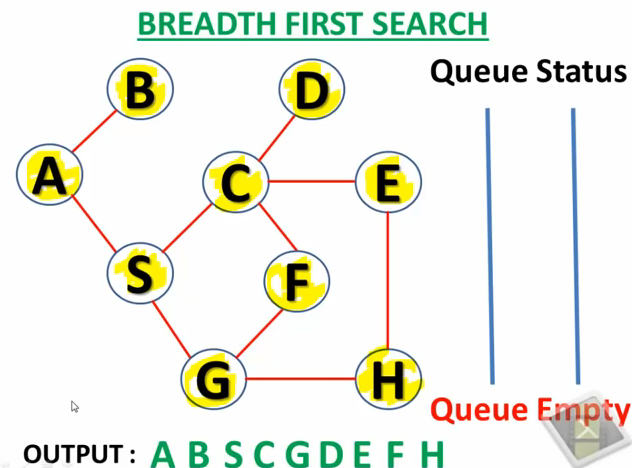
***Beispiel:***

Beginnen mit Knoten A.

Die Queue wird benötigt im alle unbesuchten Knoten zwischens zu speichern.

https://www.youtube.com/watch?v=QRq6p9s8NVg

      🡪 Nacheinander dequeuen 

**Tiefensuche:**

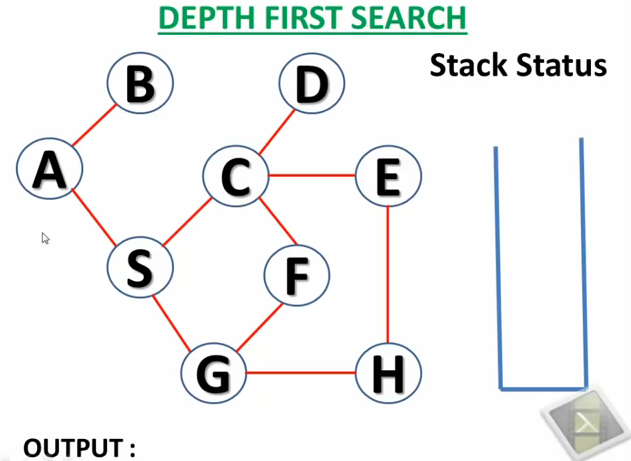
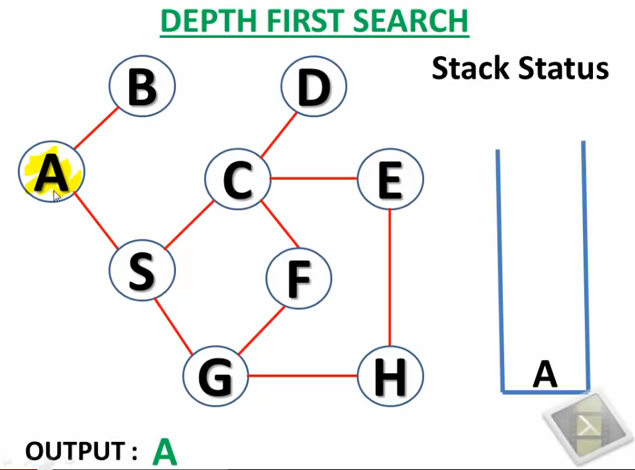
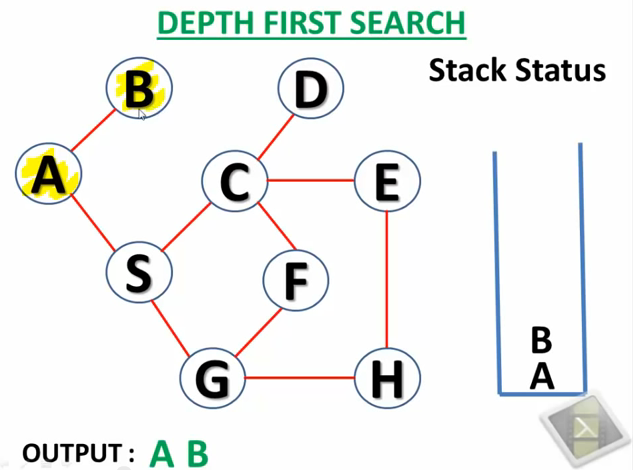
Wird zum durchlaufen des Graphen benötigt.

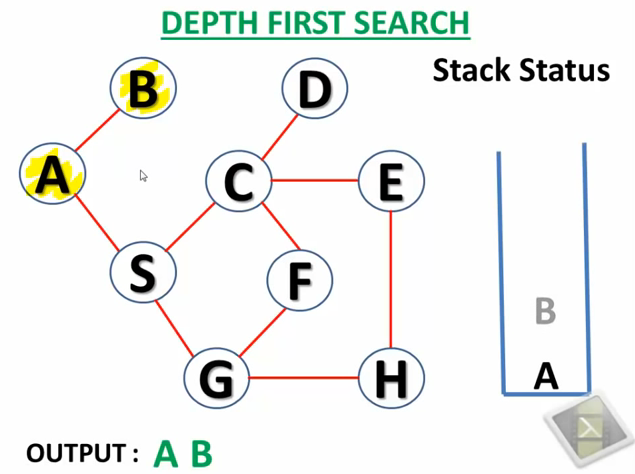
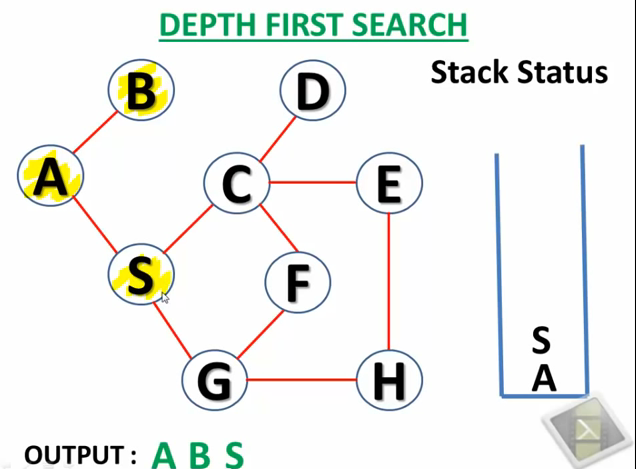
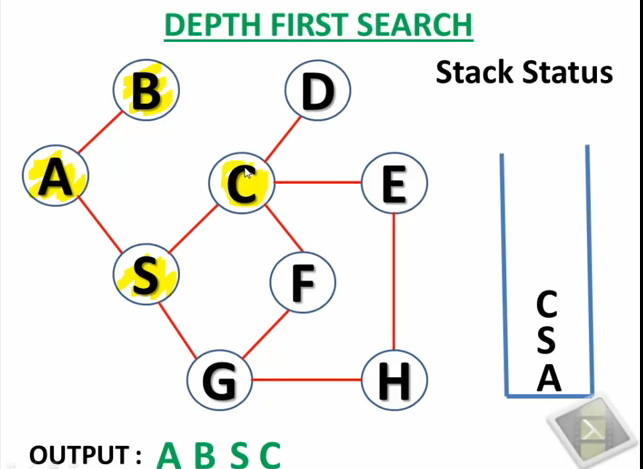
***Beispiel:***

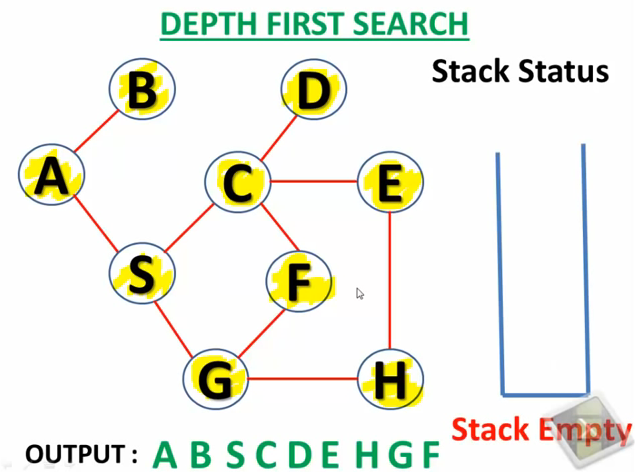
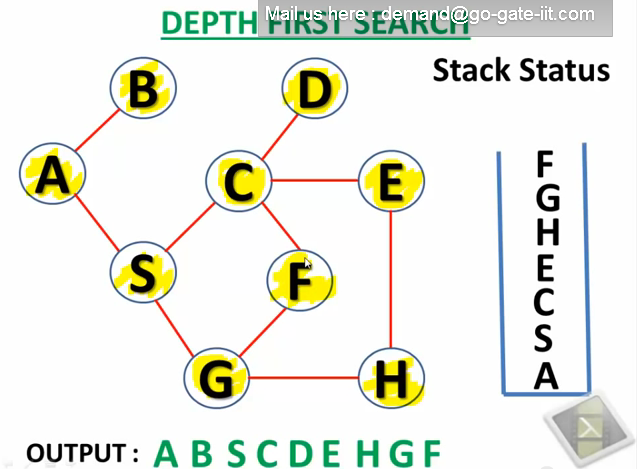
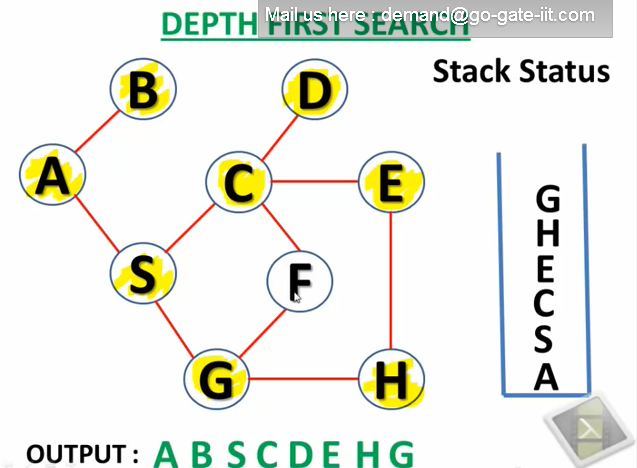
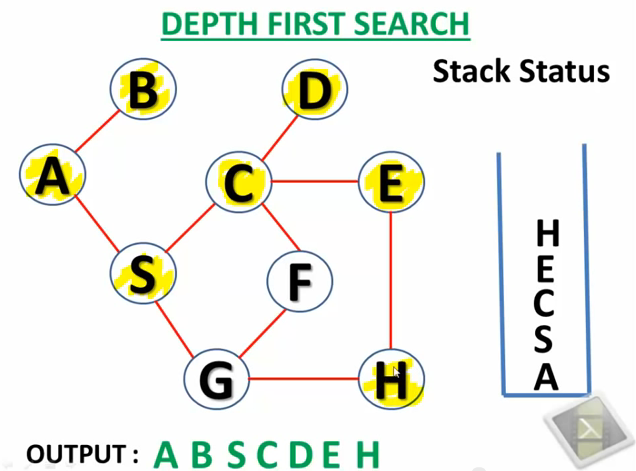
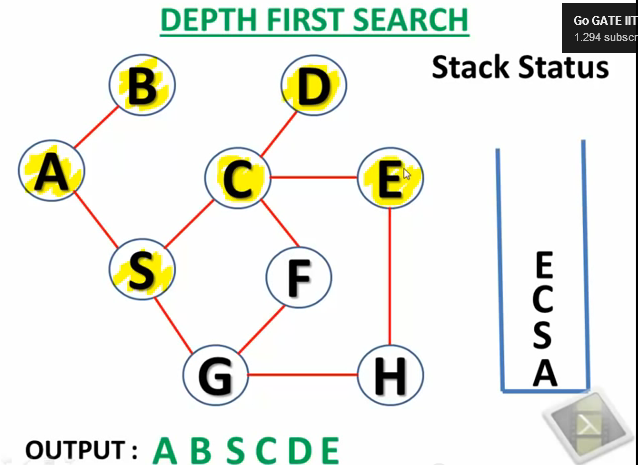
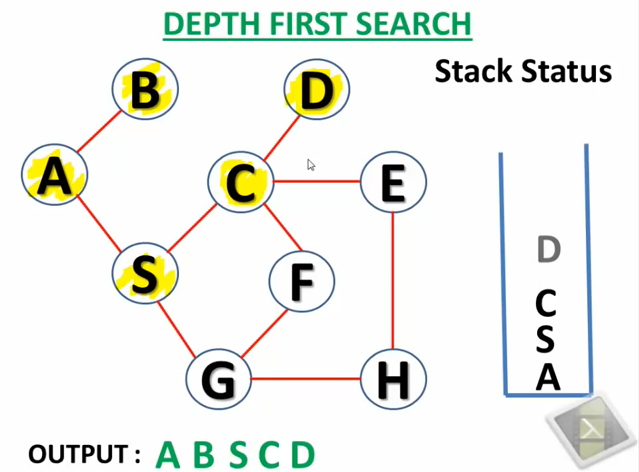
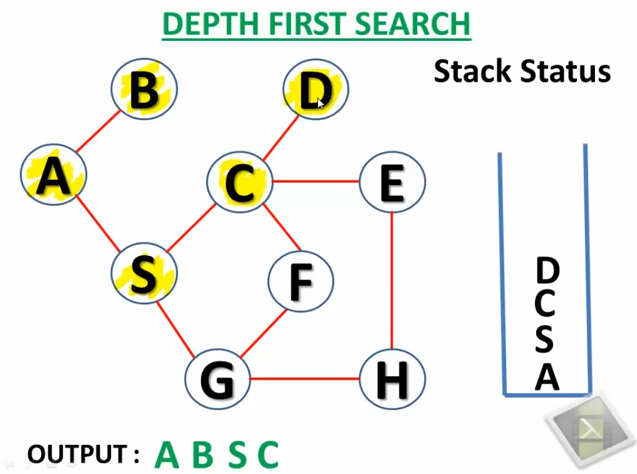
Beginnen mit Knoten A.

Der Stack wird benötigt im alle besuchten Knoten zwischens zu speichern.

https://www.youtube.com/watch?v=iaBEKo5sM7w

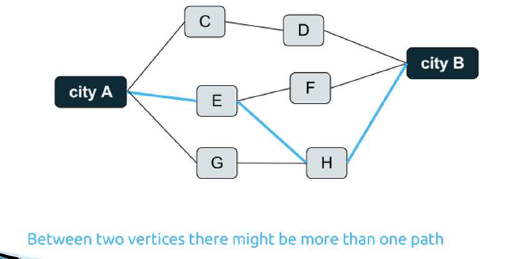


Nackeinander den Stack leeren



**Kurzester Weg von A nach B**

In ungewichteten Graphen zählt nur die Anzahl von Kanten zwischen A und B.



In gewichteten Graphen haben Kanten auch Werte. Nicht jeder Weg ist gleich „gut“. Um sicher zu gehen das wir den besten Weg überhaupt gewählt haben gibt es einige Algorithmen.

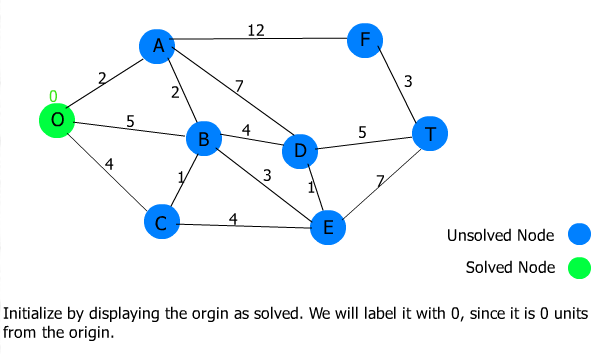
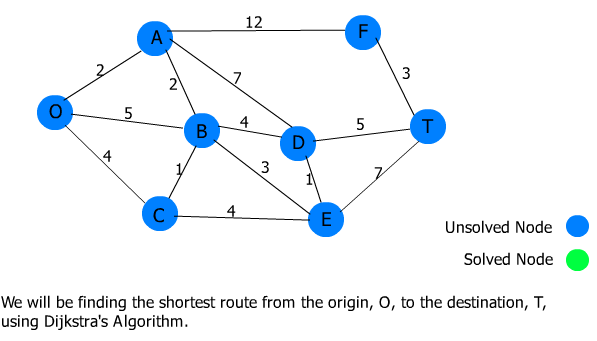
**Dijkstra Algorithmus**

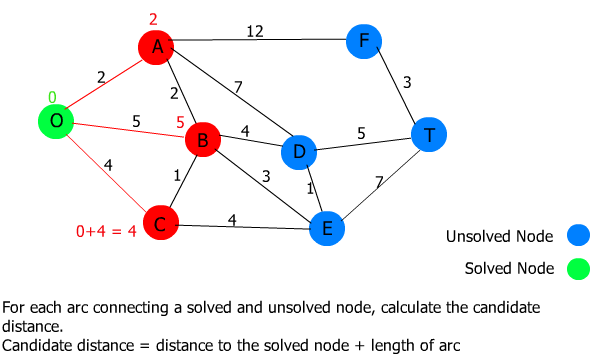
Um den besten Weg herauszufinden.

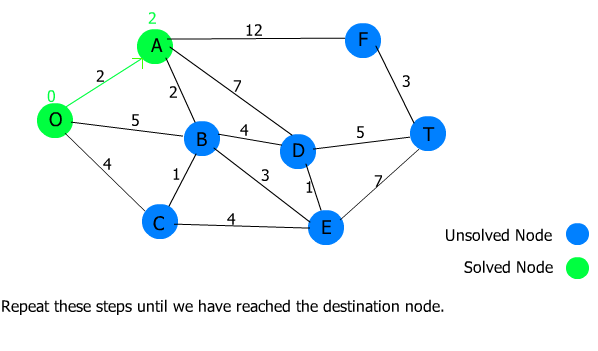
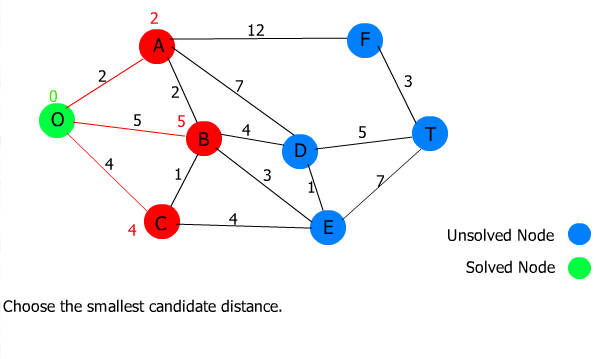
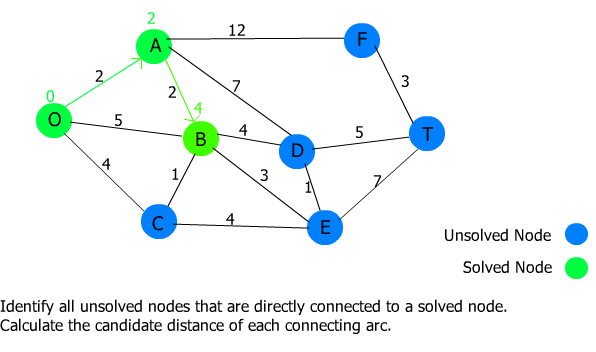
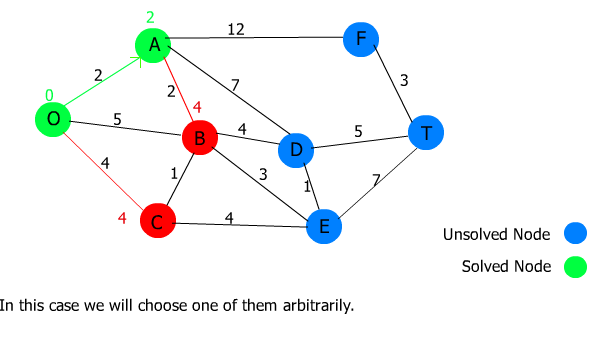
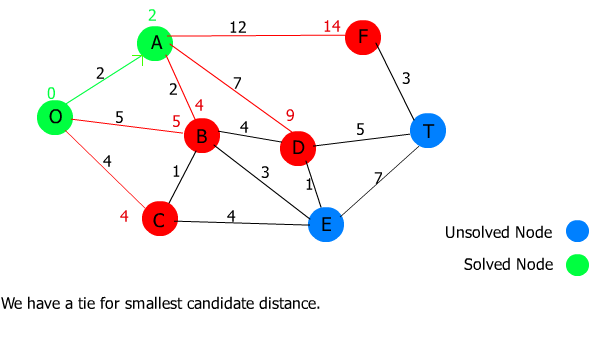
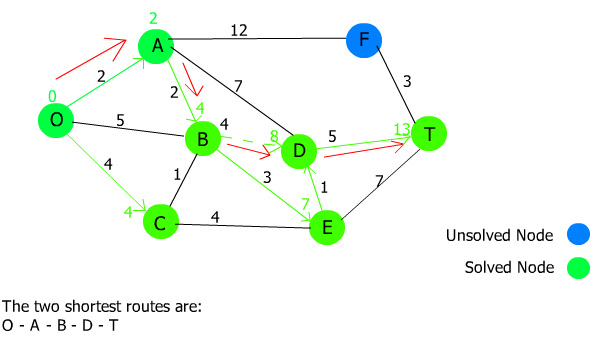
http://optlab-server.sce.carleton.ca/POAnimations2007/DijkstrasAlgo.html

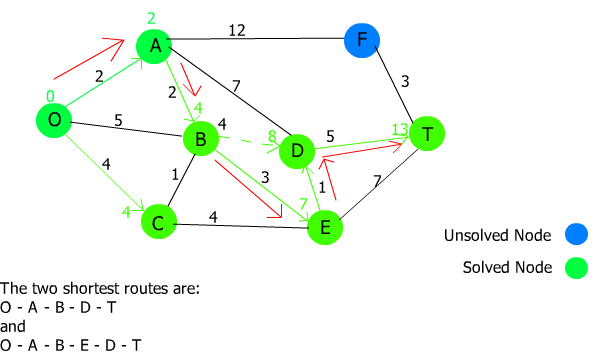
***Beispiel:***

Kürzesten Weg von A nach O bestimmen





  
usw… 🡪 Lösungen 

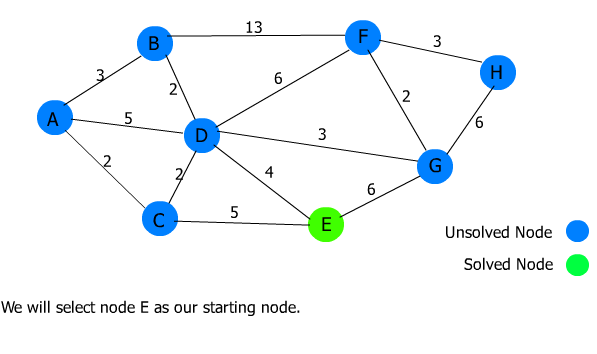
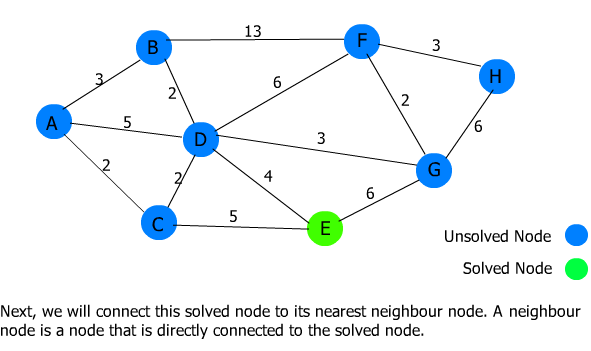


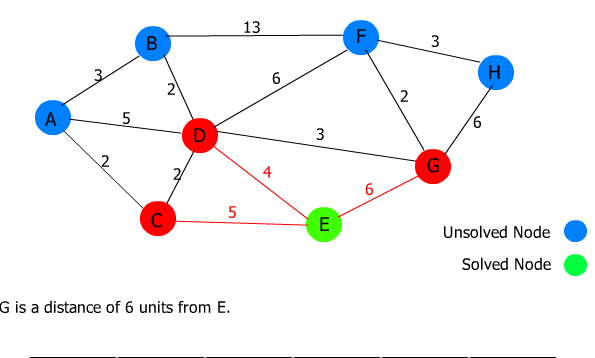
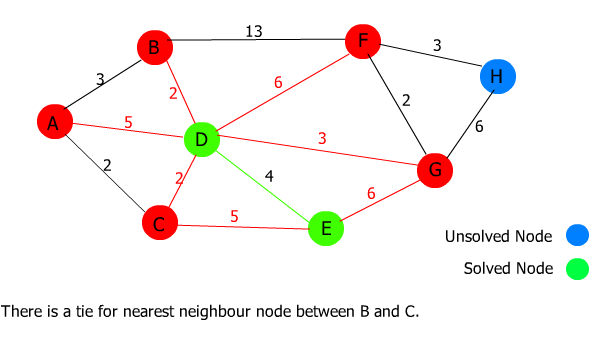
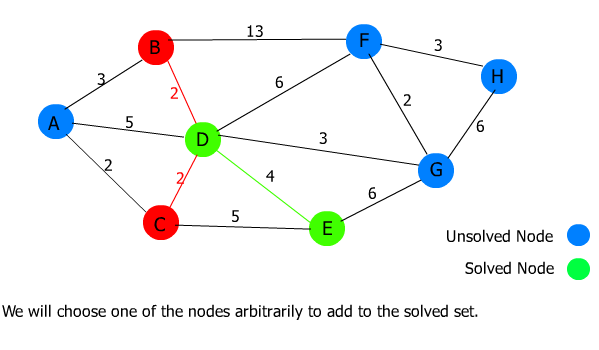
**Minimaler Spannbaum**

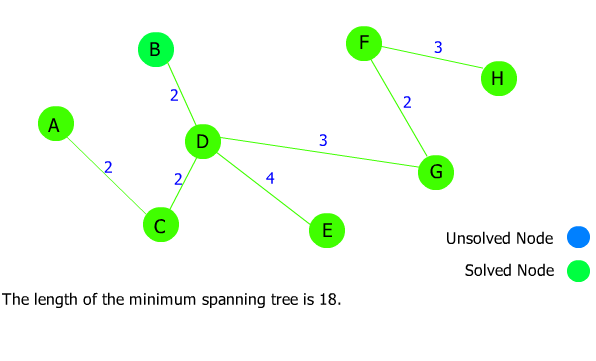
Ein minimaler Spannbaum eines Graphen ist jener Baum der alle Knoten mit der kürzesten möglichen Gesamtlänge von Kanten verbindet. ( Weil ein Baum entsteh darf es keine schleifen geben)

http://optlab-server.sce.carleton.ca/POAnimations2007/MinSpanTree.html

***Beispiel: (Jarnik Prim Algorithmus)***

… 🡪 Löung: 

**Beispiel Kruskal Algorithmus**

Ergebnis ist ein Minimaler Spannbaum (O(E log E)

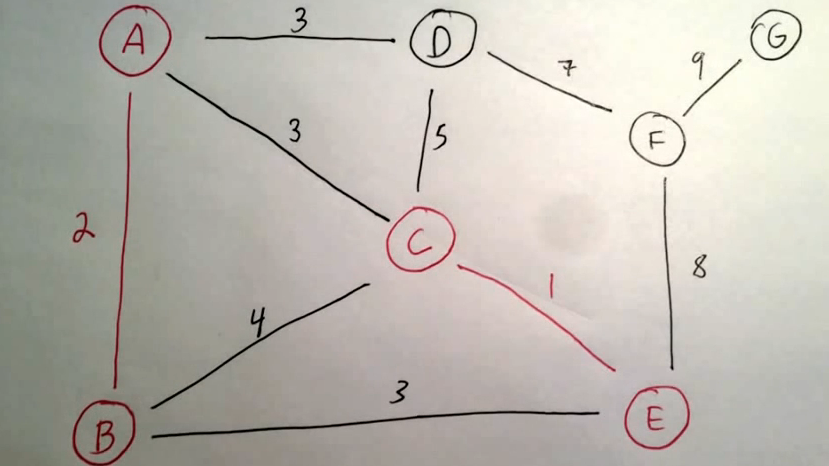
https://www.youtube.com/watch?v=71UQH7Pr9kU

***Beispiel:***

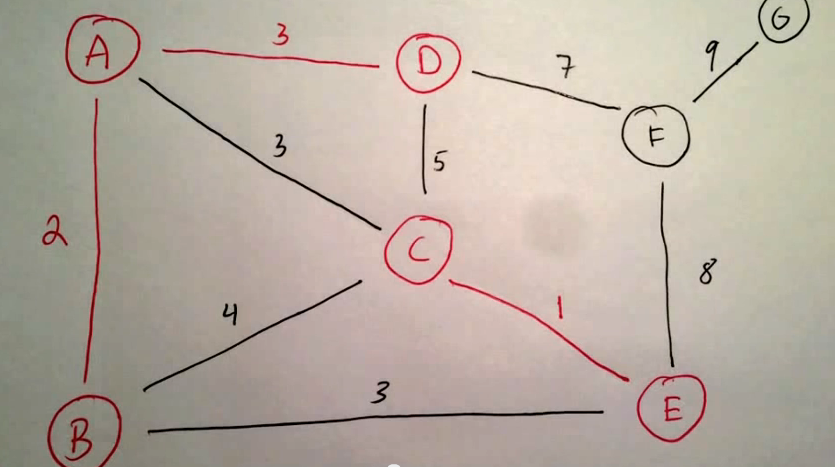
Zu beginn wähle die kleinste Kante



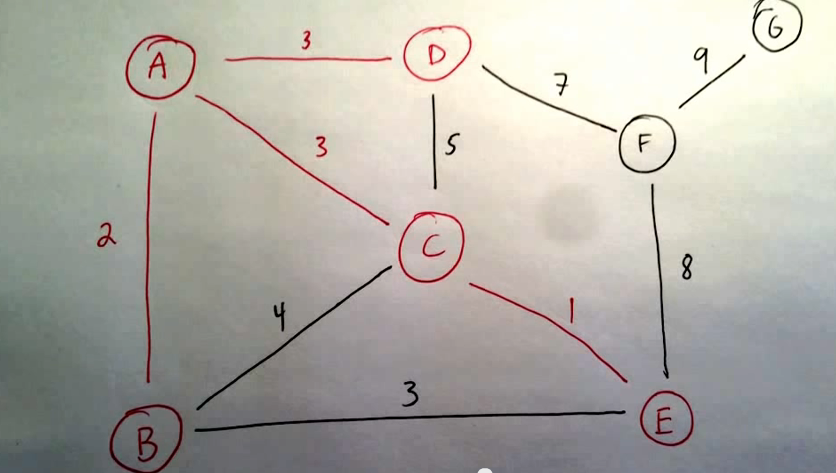
Wähle die nächste kleinste Kante die keinen Kreis bildet



Wähle die nächste kleinste Kante die keinen Kreis bildet. Im nächsten Punkt gibt es drei kanten mit der Gewichtung von 3, das heißt wir können einfach eine Auswählen



Im nächsten Punkt wählen wir die nächste Kante aus, die unterste Kante mit der Gewichtung 3 dürfen wir nun nicht mehr auswählen da wir sonst einen Kreis bilden würden.



Wenn wir alle Schritte wie bisher wiederholen kommen wir zum minimal spanning tree

