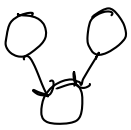
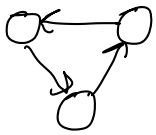
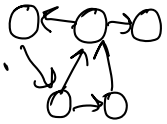


Knoten mit mehreren Kanten



Kreis

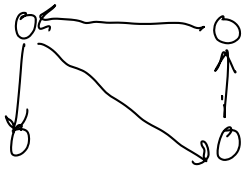


Beliebig Komplex

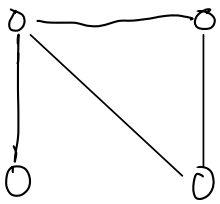
Begriffe:

- Knoten, Kanten
- Pfad
- Zyklus

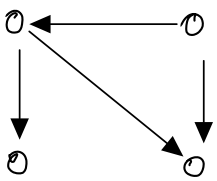
gerichtete Graphen



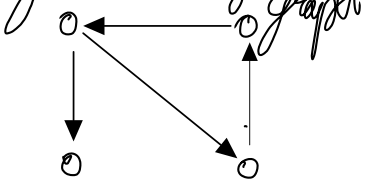
ungerichtete Graphen



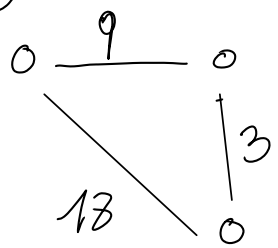
azyklischer gerichteter Graph



gerichteter zyklischer Graph



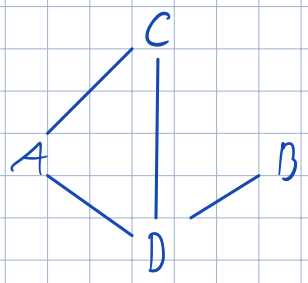
gewichtete Graphen



Grapen ≠ Bäume

Bäume = Graphen

Speicherdarstellung



Adjazenzliste

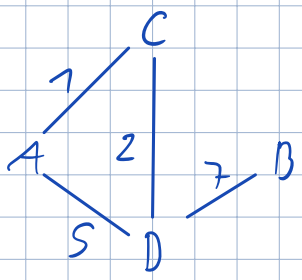
```
class Node {
  String value;
  ArrayList<Node> childs;
  ⋮
}
```

Growable Array

Adjazenzmatrix

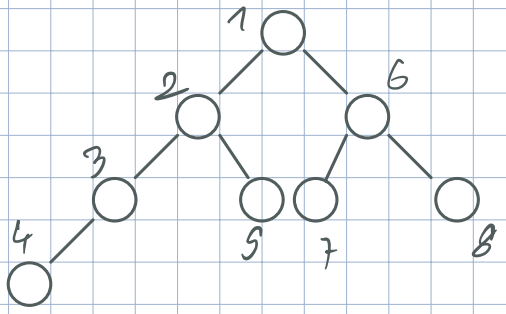
```
boolean[][] adj;
```

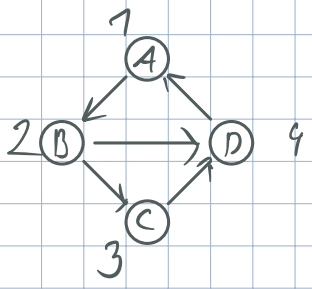
	A	B	C	D
A		0	1	1
B	0		0	1
C	1	0		1
D	1	1	1	



	A	B	C	D
A		0	1	5
B	0		0	7
C	1	0		2
D	5	7	2	

Depth-First-Search





Naive Idee:

```
void visit (Node p) {
  ... process p ...
  for all (sons s of p) visit(s);
}
```

⇒ Problem Zyklen:

```
class Node {
  ... val;
  boolean marked;   // true == schon besucht
  Node[] sons;
}

void DFS (Node p) {
  p.marked = true;
  ... process p ...
  for (Node son : p.sons)
    if (!son.marked) DFS(son);
}
```

O(n)

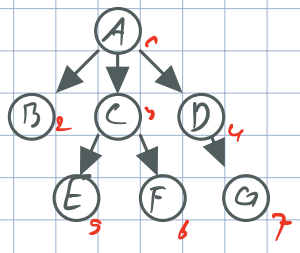
Rücksetzen der Markierung

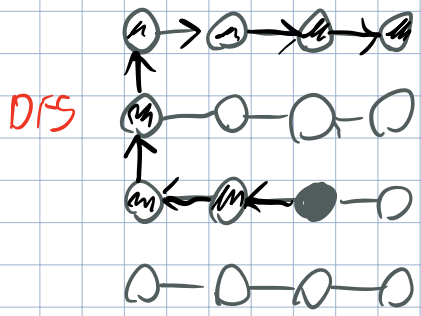
→ nochmals durchlaufen

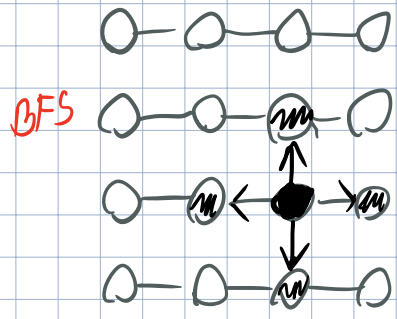
```
void DFS (Node p) {
  p.marked != p.marked;
  ... process p ...
  for (Node son : p.sons)
    if (son.marked != p.marked) DFS(son);
}
```

Laufzeit: Problemgröße: Anzahl der Kanten $n \rightarrow O(n)$

Breadth-First-Search







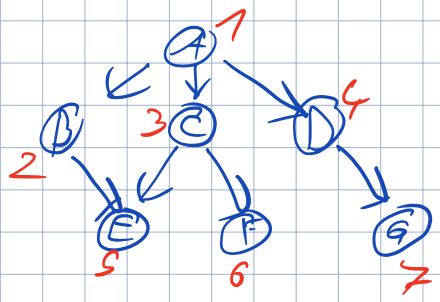
O(n)

```
void BFS (Node p){

    Queue q = new Queue ();
    p.marked = true;

    do{

        ..... process p .....
        for (Node son : p.sons )

            if(! son.marked ){
                q.put (son);
                son. marked = true;
            }

        }
        p = q.get();
    } while (p != null );
}
```

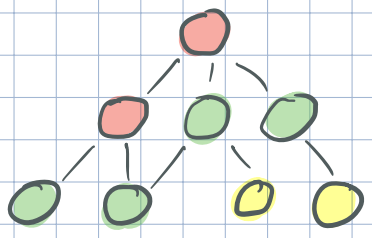
Kurzschreibweise um über ein Array zu iterieren

gibt null zurück, falls die Queue leer ist

Besuch von	Queue		
A	B	C	D
B	C	D	E
C	D	E	F
D	E	F	G
E	F	G	
F	G		
G			



Allgemeines Schema für die Graf Traversierung



Aufteilen in 3. Gruppen

Baumknoten: besuchte Knoten

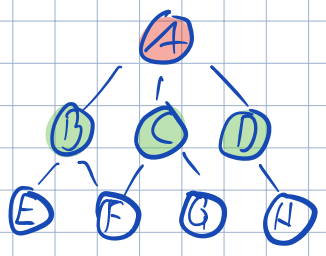
Saumknoten: unbesuchte Nachbarn von Baumknoten

ungesehene Kind:

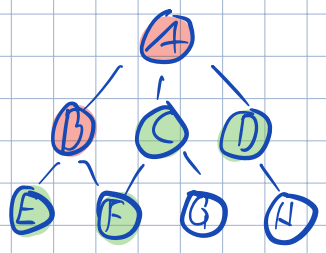
Traverse:

```
Baumknoten = { }
Saumknoten = {Wurzel}

while ( Saumknoten != {} ){
    • Mache einen Saumknoten k zu einem Baumknoten
    • Mache alle ungesehenen Nachbarn von k zu Saumknoten
}
```

BFS



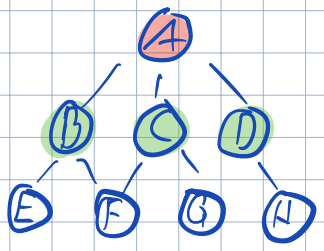
Saum: B C D



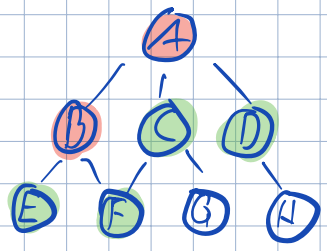
Saum: C D E F

⇒ Saum Datenstrukturen ist eine Queue

DFS



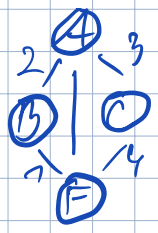
Saum: D C B



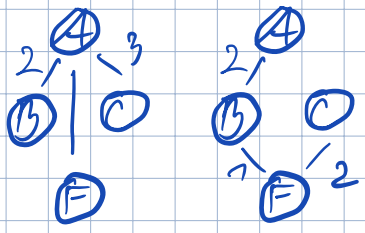
Saum: D C F E

Saum Datenstruktur ist ein Stack

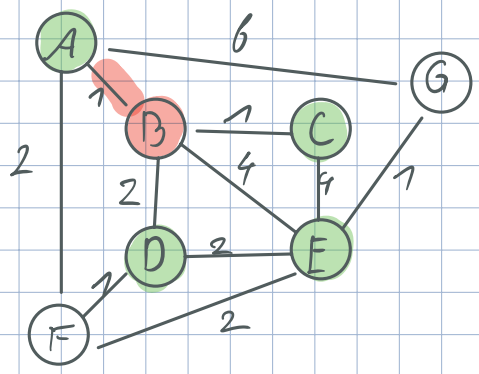
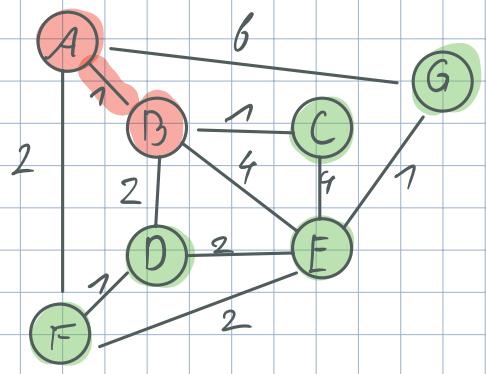
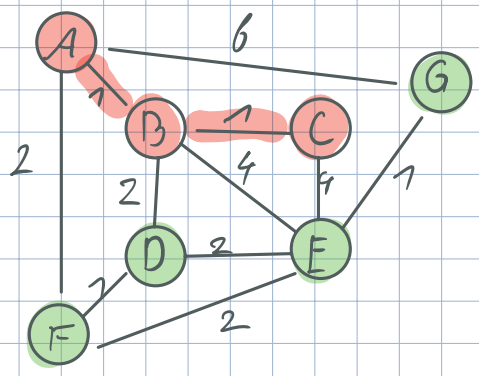
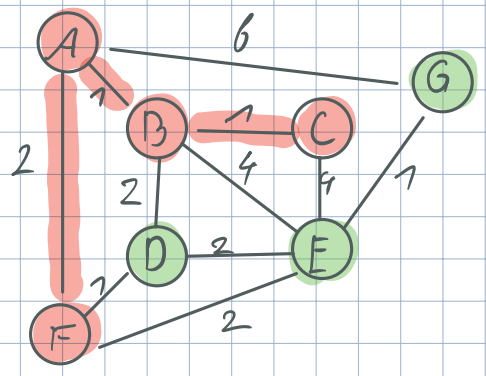
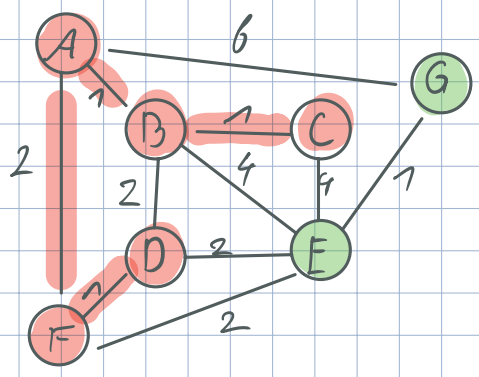
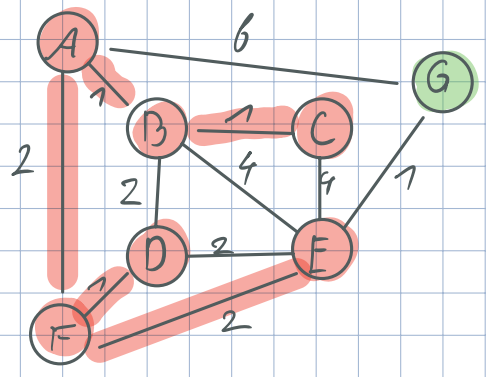
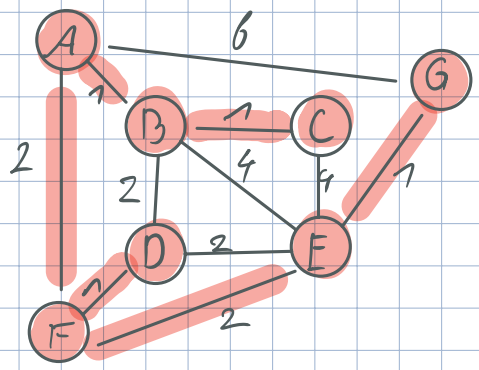
Minimal Spanning Tree



Spannender Baum: Baum der alle Knoten eines Graphen enthält



Kleinster Spannender Baum: Baum mit dem geringsten Gewicht

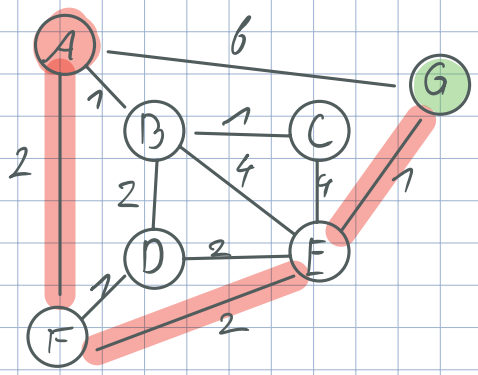
p = Wurzel;
Baumknoten = {p}; Saumknoten = { };

while (noch nicht alle Knoten sind Baumknoten) {

  Mache alle ungesehenen Nachbarn von p zu Saumknoten
  p = Saumknoten mit kleinstem Abstand zu irgendeinem Baumknoten d;
  nimm Kante dad → p in MST auf;
  mache p zu Baumknoten;
}

- Mache einen Saumknoten k zu einem Baumknoten

Shortest Path



Baum	dad	Saum
A	/	(B, A|1), (F, A|2), (G, A|6)
B	A	(F, A|2), (C, B|2), (D, B|3)
		(E, B|5), (G, A|6)
F	A	(C, B|2), (D, B|3), (E, F|4)
		(G, A|6)
C	B	(D, B|3), (E, F|4), (G, 6|A)
D	B	(E, F|4), (G, A|6)
E	F	(G, E|E),
G	E	/

Vater
Distanz
AB+DC
Priorität ordnet sich

A - F - E - G

Idee kürzeste Verbindung von A → B

Baumknoten = {A}

Saumknoten = {Nachbarn von A}

while (B nicht im Baumknoten){

q = Saumknoten mit kürzester Distanz zu A

mache q zu Baumknoten und ungelesene Nachbarn zu Saumknoten

}

wichtig!

```
void ShortestPath (Node p, Node q) {  // find shortest path from p to q
   Heap heap = new Heap();
   p.distance = 0;
   while (p != q) {
      p.marked = true;
      for (int i = 0; i < p.son.length; i++) {
         Node son = p.son[i];
         if (!son.marked) { // for all sons
            int distance = p.distance + p.weight[i];
            if (son.pos == 0) { // not yet in PQ
               son.dad = p; son.distance = distance;
               heap.insert(son);
            } else if (distance < son.distance) { // change position of son in PQ
               son.dad = p; son.distance = distance;
               heap.upHeap(son.pos);
            }
         }
      } // for all sons
      p = heap.remove();
   }
}
```

nicht auswendig lernen