Grafer

I. Graf

- **definisjon**
- **terminologi**
- **noen eksempler på graf-problemer**

II. Graf Traversering

- **DFS**
- **BFS**

III. Graf ADT og Implementasjon

- **Kant-Liste og Nabo-Liste**
- **Nabo-Matrise**

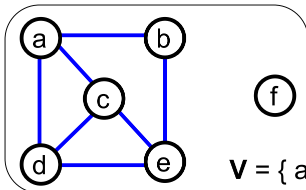
IV. Rettede Grafer (diGraphs)

V. Vektede Grafer

Kap. 9 (kursorisk 9.4.5)
Kap.10 (untatt 10.2.2–10.2.4, 10.3)

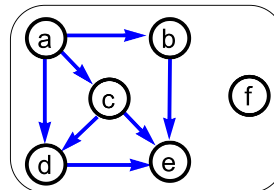
En graf G

er gitt ved to mengder (E,V)
V av noder
E av kanter
der en kant e ε E er
et (uordnet) par {u,v} av noder u, v ε V

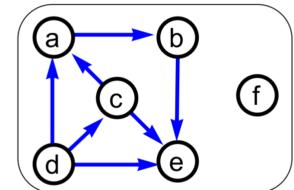


V = { a, b, c, d, e, f }
E = { (a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e) }
= { (b,a), (c,a), (d,a), (e,b), (d,c), (e,c), (e,d) }

En **rettet graf** (diGraf)
V av noder
E av kanter
der en kant e ε E er
et ordnet par (u,v), u, v ε V : u → v



E = { (a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e) }



E = { (a,b), **(c,a)**, **(d,s)**, (b,e), **(d,c)**, (c,e), (d,e) }

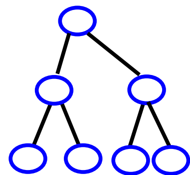
En **ikke-rettet graf** er en (rettet) graf der relasjonen E er **symmetrisk**
u → v ε E *hviss* v → u ε E.

En (rettet) **graf** er gitt ved
en mengde V av noder
en **binær relasjon** E ⊆ V × V

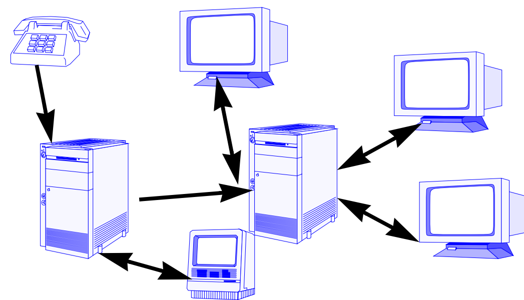
Anvendelser ...

I. Binære relasjoner

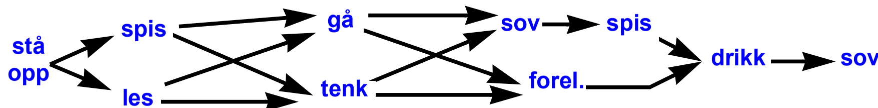
II. Trær



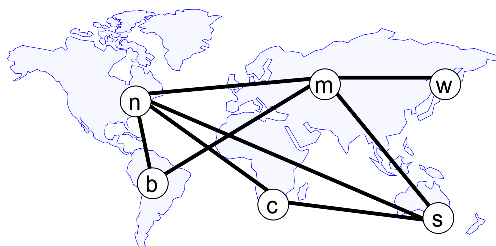
III. Nettverk

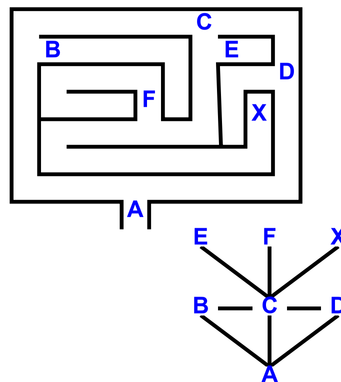


IV. Planlegging



V. Forbindelser





. . .

Graf terminologi

nabo noder : forbundet med en kant a-c, c-e

graden til en node : antall nabo noder (kanter) deg(c)=3, deg(f)=0

$$\sum_{v \in V} deg(v) = 2(\#\ \text{kanter})$$

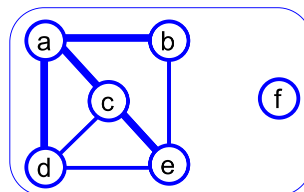
inngrad/utgrad : antall innkommende/utgående kanter (rettede grafer)

sti : en sekvens n_1, n_2 ... n_k av noder slik at $(n_i, n_{i+1}) \varepsilon E$

acaca, bec, abedc, edce

enkel sti : ingen node forekommer 2 ganger

sykel : enkel sti men $n_1 = n_k$

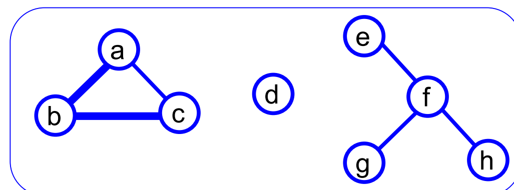


sammenhengende graf : det finnes en sti mellom alle par av noder

delgraf : en delmengde av V og E som er en graf

sammenhengende komponent : en maksimal sammenhengende delgraf

komplett graf : for hver par av noder u, v ε V, finnes det en kant (u,v) ε E



(ikke-rotet) **tre** : sammenhengende graf uten sykler

utspennende tre for en graf G : en delgraf av G som er et tre og inneholder alle G's noder

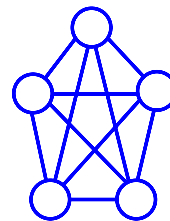
skog : samling av trær

DAG :rettet graf uten sykler (directed acyclic graph)

1. Telling av noder og kanter

1.a) Litt om sammenhenger...

n = # noder, ***k*** = # kanter



n =5
k =10

- G er **komplett hviss** hver node har (***n*** –1) naboer

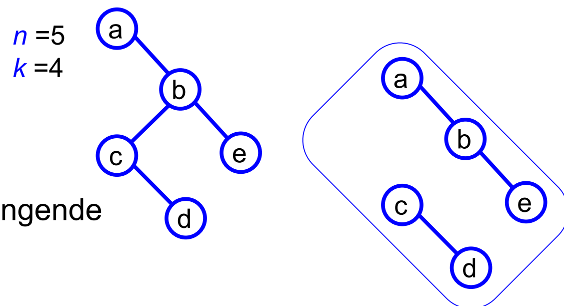
  dvs. ***k*** **=** 1/2 * $\sum_{v \in V} deg(v)$ **=** 1/2 * ***n*** * (***n*** –1)

- G **ikke komplett hviss** ***k*** **<** 1/2 * ***n*** * (***n*** –1)

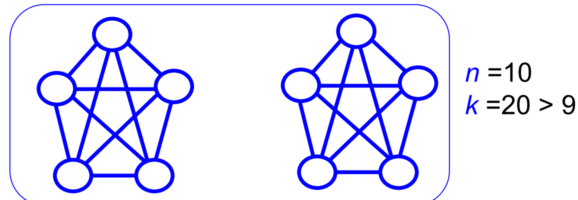
- **Hvis** G **er et tre så** ***k*** **=** ***n*** –1

  - et minimalt antall kanter som kan gi en sammenhengende graf av ***n*** noder
  - fjerning av en vilkårlig kant, gjør grafen usammenhengende
  - G trenger ikke å være et tre selv om ***k*** **=** ***n*** –1

n =5
k =4

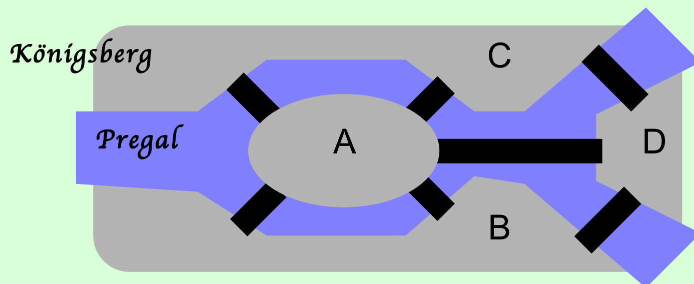


- **Hvis** ***k*** **<** ***n*** –1 **så er ikke** G **sammenhengende**
  men ikke omvendt !!!

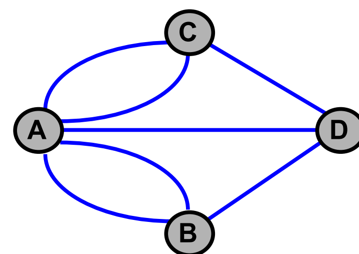


n =10
k =20 > 9

1.b) *Kants spasering og Eulers tur*



Tillatt grafer med flere kanter mellom to noder. (multigrafer)



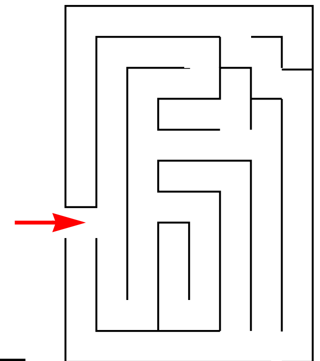
Eulers tur : en sti som traverserer hver kant nøyaktig én gang

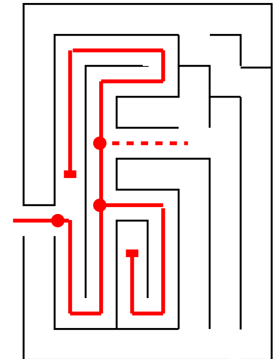
Eulers teorem : En (sammenhengende) graf G har en Eulers tur hvis og bare hvis G har 0 eller 2 odde noder (graden er oddetall) **O(n+k)**

Hamiltonsk sykel : en enkel sykel som traverserer hver node nøyaktig en gang **O(n!)**

... **...**

2. *Ariadnes (t)råd*





Minos :
OK, Theseus, but you must first find and kill Minotaur hidding in the Labirynth.

Theseus :
I'm good at killing but how the heck am I going to find it and then get out of there **?**

Ariadne :
(she felt in love with Theseus in the meantime...)

Ta denne tråden og fest den ved inngangen til Labirynten.

(k) Hver gang du kommer til et nytt kryss **u**, merk **u** '**visited**'.

(r) Velg en vilkårlig vei – merk den og følg men **dra tråden** etter deg

Når du så kommer til et nytt kryss **v** : dersom det er
- en blind gate, gå tilbake **langs tråden** (**nøst** den igjen)
- et kryss merket '**visited**', gå tilbake **langs tråden** (**nøst** den igjen)
- et umerket kryss, **gjenta** det hele (k)

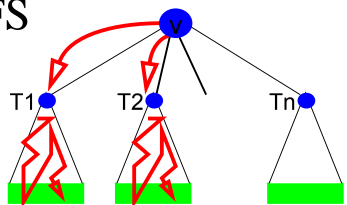
Etter hvert vil alle veier (r) fra krysset **u** bli merket
- gå da tilbake **langs tråden** til forrige krysset og fortsett å utforske umerkede veier derfra.

Du vil på denne måten kunne utforske hele labirynten og returnere til inngangen

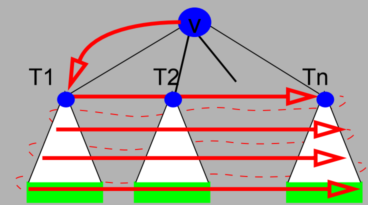
Chorus :
Smart Girl!

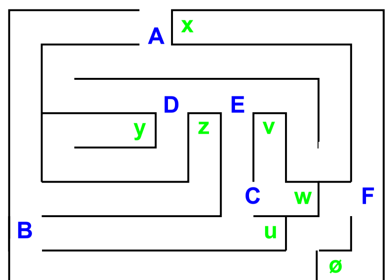
2. Graf traversering

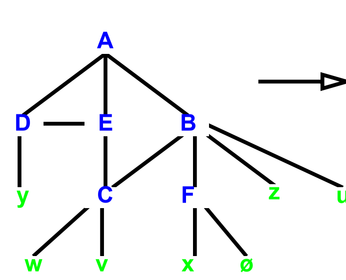
a) DFS



b) BFS



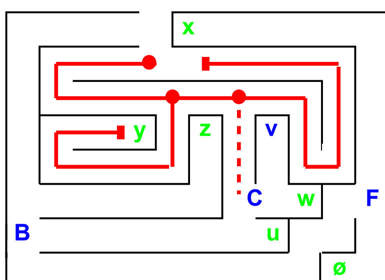





```
DFS(A)
  merk A visited
  for hver kant e = (A, X)
    if ( ! merket X )  // ! e er rød
      // merk–e–rød
      DFS(X)
    // else merk–e–svart
```

Kanter merket med rød under DFS traversering gir et **utspennende tre for grafen**
– roten til treet kan velges vilkårlig !

A
D E B
y C F z u
w v x ø



2.a) DFS traversering av grafer

```
DFS(s)
  merk s visited
  for hver kant e = (s,u)
     if ( ! merket–u )
        // merk–e–rød
        DFS(u)
```

9.12 DFS traversering av en ikke-rettet graf G fra en node s:

a) besøker *alle* noder i en *sammenhengende komponent* til s

b) merkete kanter gir et *utspennende tre*, DFS tre, for denne komponenten til s

Begrunnelse :

a) Anta, kontrapositivt, at det finnes en ubesøkt node v.
Siden komponenten er sammenhengende, finnes det en sti fra s til enhver node,
så anta at v er den første ubesøkte noden på en sti fra s:

- Siden v er den første slike, finnes det en nabo node u ("like før") som ble besøkt
- Men da – mens vi besøkte u – måtte vi også ha sett på kanten (u,v) og, siden v ikke var merket, måtte den ha blitt besøkt.

b) Vi merker kanter (u,v) kun når vi går til endenoden v for første gang

- Derfor danner vi aldri en sykel – vi fåren asyklisk graf, dvs. et tre
- Treet er utspennende fordi alle komponentens noder er med (a)

2.a) DFS traversering av grafer


```
DFS(s)
  merk s visited
  for hver kant e = (s,u)
     if ( ! merket–u )
        // merk–e–rød
        DFS(u)
```


Kjøretid av DFS:

DFS kalles 1 gang for hver node og ser hver kant 2 ganger :
$O(n_s + k_s)$ hvis

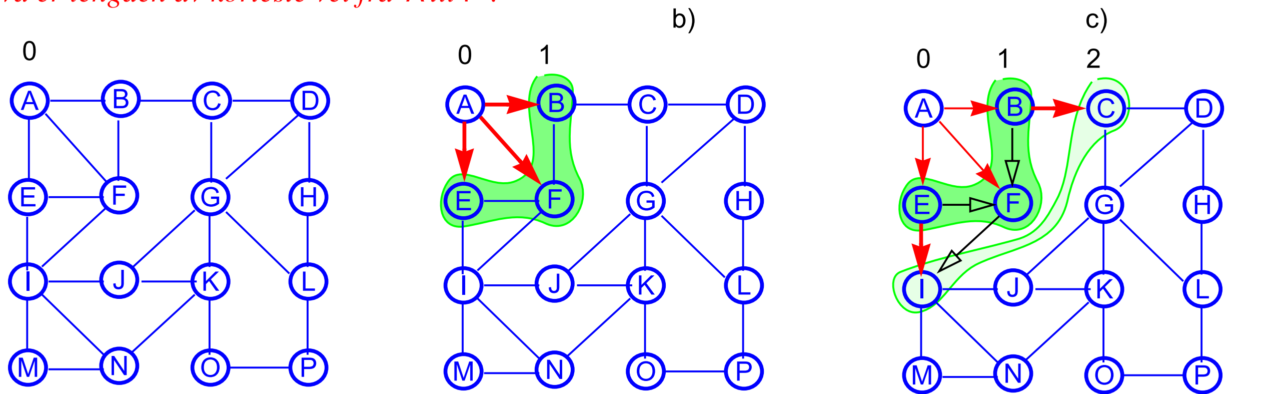
- gitt en kant, kan man aksessere dens ende-noder i $O(1)$
- merking av noder/kanter og sjekking om de er merket tar $O(1)$
- for hver node v, kan alle dens kanter akssesere 1 gang i $O(k(v))$

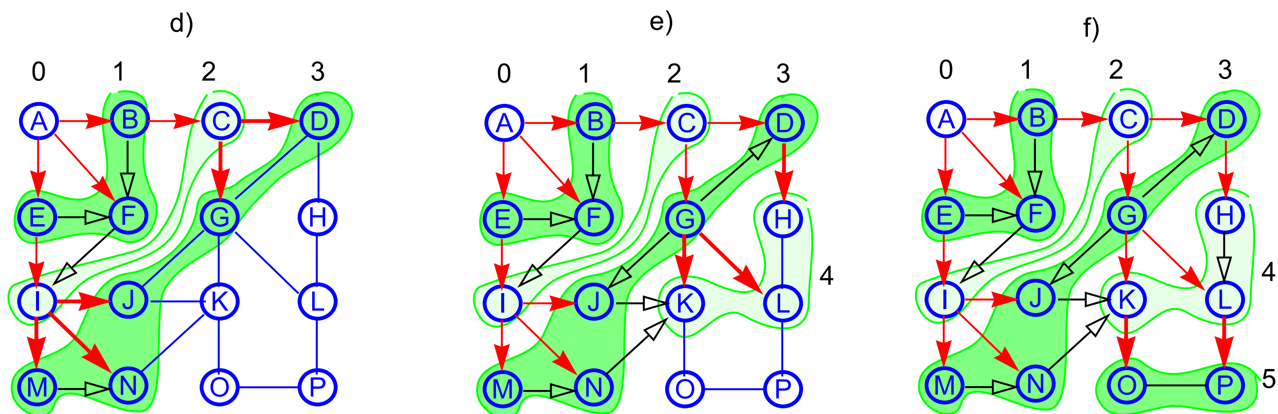
*DFS kan brukes for å lage **O(n+k)** algoritmer for å :*

- sjekke om G er sammenhengende og finne sammenhengende komponenter
- finne utspennende tre for G
- sjekke om det finnes en sti mellom to noder;
- se om G inneholder sykler

2.b) BFS traversering av grafer

Hva er lengden av korteste vei fra A *til* P *?*





2.b) BFS graf traversering

Tree DFS

```
/*DFS(Tree T,Position s)
 *   Stack S = new StackIM()
 *   S.push(s)
 *   while (!S.isEmpty())
 *     p= S.pop()
 *     for each p's child c
 *        S.push(c)
 */
```

Tree BFS

```
/*BFS(Tree T,Position s)
 *   Queue S = new QueueIM()
 *   S.enqueue(s)
 *   while (!S.isEmpty())
 *     p= S.dequeue()
 *     for each p's child c
 *        S.enqueue(c)
 */
```

Graph BFS

```
// initielt er alle noder umerket

BFS(Graph G, Position s)
   Queue S = new QueueIM()
   merk(v, 0) – merker med nivå
   S.enqueue(s)
   while (!S.isEmpty())
     k= S.dequeue()
     for each kant e=(k,m)
       if (! merket(m))
          merk(m, k.merke+1)
          merk(e, rød)
          S.enqueue(m)
```

BFS gir opphav til O**(n+k)** algoritmer for å

- sjekke om G er sammenhengende
- finne sammenhengende komponenter i G
- finne utspennende tre for G
- beregne minimalt antall kanter mellom to noder (korteste sti)

9.14. BFS traversering av en ikke-rettet graf G fra en node s:

a) besøker alle noder i en sammenhengende komponent til s

b) røde kanter danner et utspennende tre, så kalt BFS tre, for G

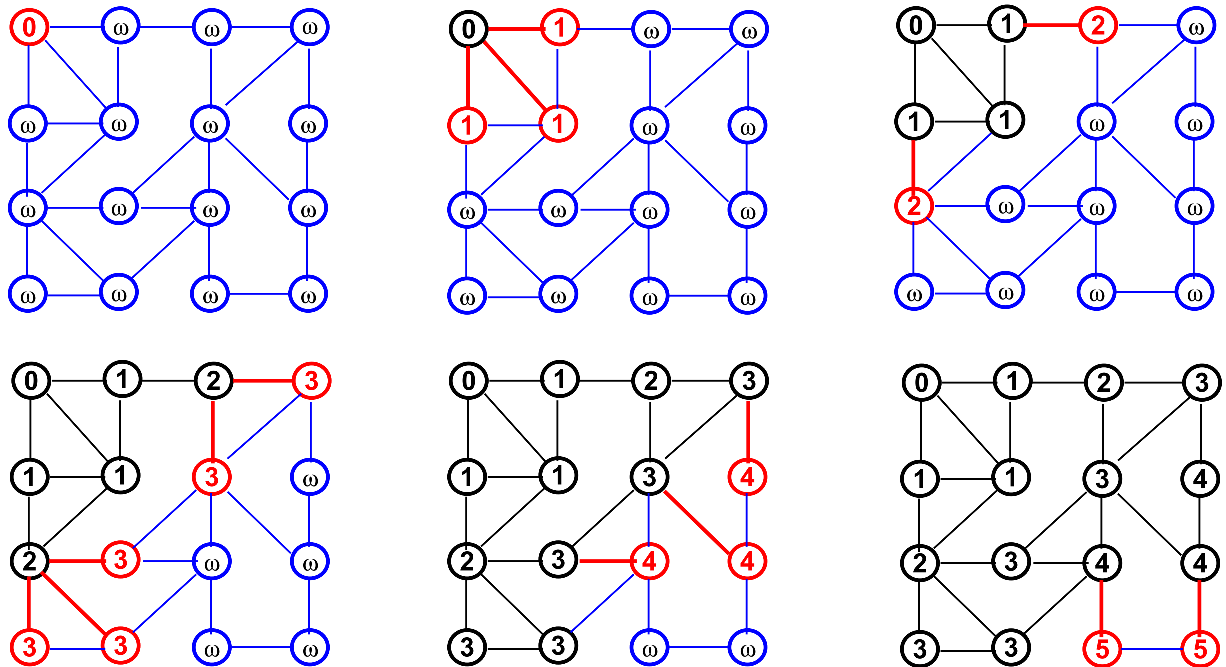
c) korteste stien fra **s** til hver node på nivå ***i*** har ***i*** kanter og enhver annen sti har minst ***i*** kanter

d) er ikke kant **(u,v)** med i BFS tre, så er nivå forskjellen mellom **u** og **v** høyst ***1***

BFS: Ford-Bellman

```
for each node v : D[v]= ω          // ω er et maksimalt tall (her ω > n)
D[s] = 0;                          // s er startnoden
for (i=1; i<n; i++)                // n er antall noder i grafen G
   for each kant (k,m)             // for en ikke-rettet G : (k,m) ε G hviss (m,k) ε G
      if ( D[k] + 1 < D[m])  D[m] = D[k] + 1
```

O(n*k)



Graf ADT *(ikke-rettet utsnitt)*

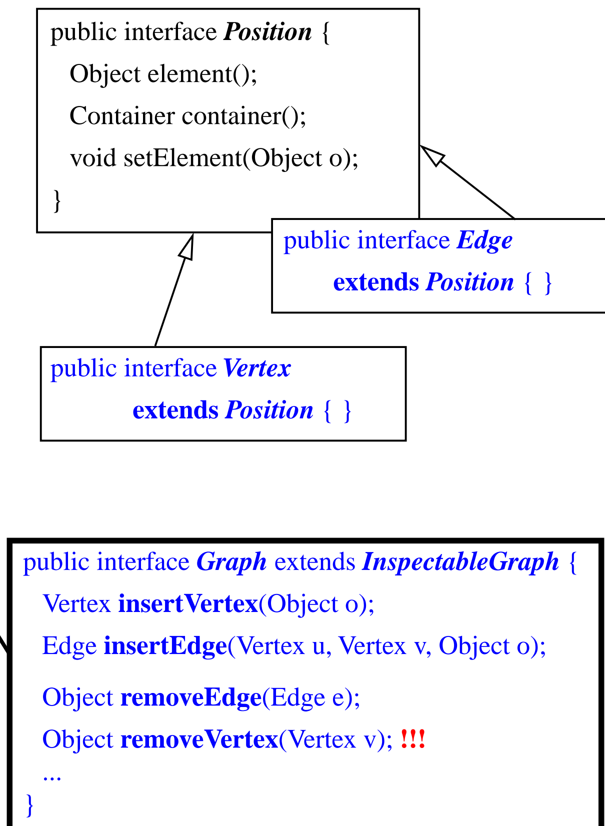
package jdsl.core.api;

```
public interface PositionalContainer
                    extends Container {
// size(); isEmpty(); elements();
  Enumeration positions();
  void swap(Position p, Position q);
  object replace(Position p, Object o);
}
```

```
public interface InspectableGraph
                    extends PositionalContainer {
   int numVertices();
   int numEdges();
   Enumeration vertices();
   Enumeration edges();

   Vertex[] endVertices(Edge e)
   Vertex opposite(Vertex v, Edge e);

   int degree(Vertex v);
   Enumeration adjacentVertices(Vertex v);
   Enumeration incidentEdges(Vertex v);
   ...
}
```



```
public interface Position {
   Object element();
   Container container();
   void setElement(Object o);
}
```

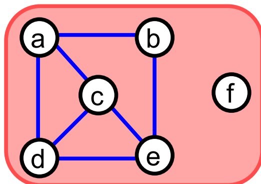
```
public interface Edge
      extends Position { }
```

```
public interface Vertex
      extends Position { }
```

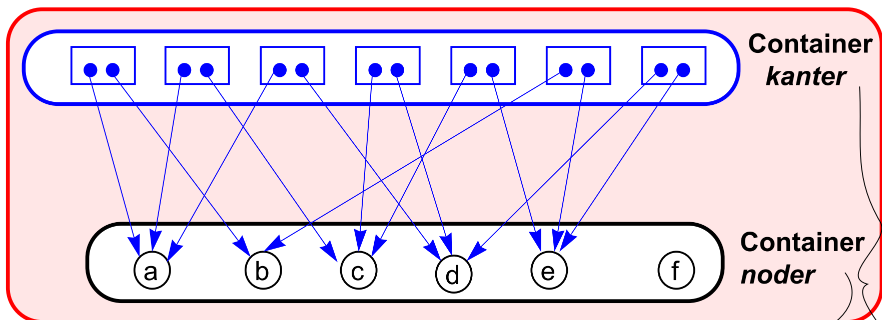
```
public interface Graph extends InspectableGraph {
   Vertex insertVertex(Object o);
   Edge insertEdge(Vertex u, Vertex v, Object o);

   Object removeEdge(Edge e);
   Object removeVertex(Vertex v); !!!
   ...
}
```

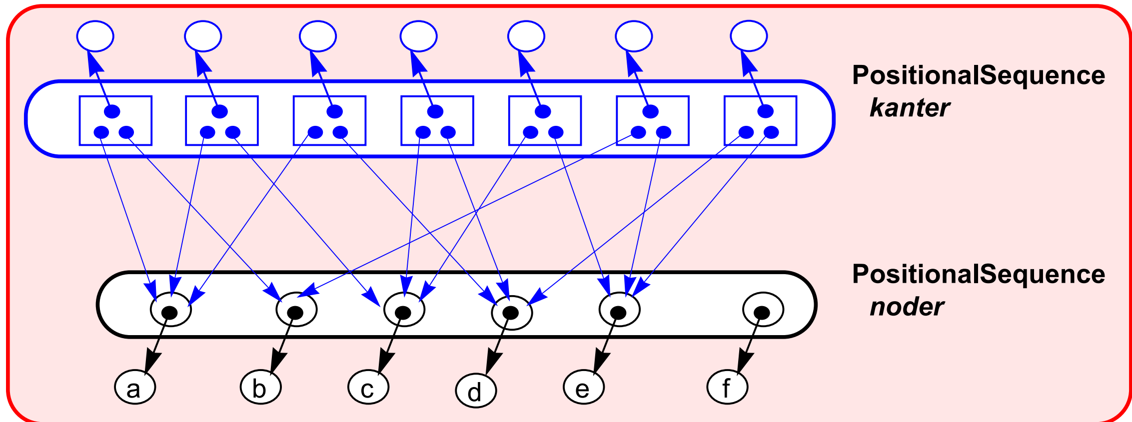
1. Implementasjon av Graph *med Kant-Liste*



endV(e), opposite(v,e),
  degree(v) .... $O(\mathbf{1})$
insertV(o), insertE(v,u,o),
  removeE(e) .... $O(\mathbf{1})$
incidentE(v), adjacentV(v) .. $O(\mathbf{k})$
removeV(v) .... $O(\mathbf{k})$
areAdjacent(v,u) .... $O(\mathbf{k})$

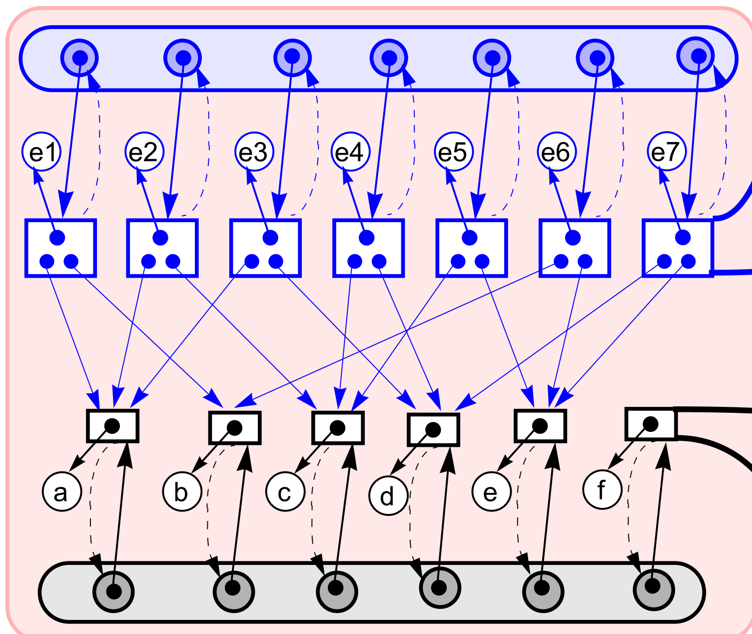


plass forbruk $O(\mathbf{k+n})$



public class *GraphKL* implements *Graph* {

private PositionalSequence **kanter**, **noder**;
public ***GraphKL***(PositionalSequence **k**, **n**)
  { **kanter** = k; **noder** = n; }



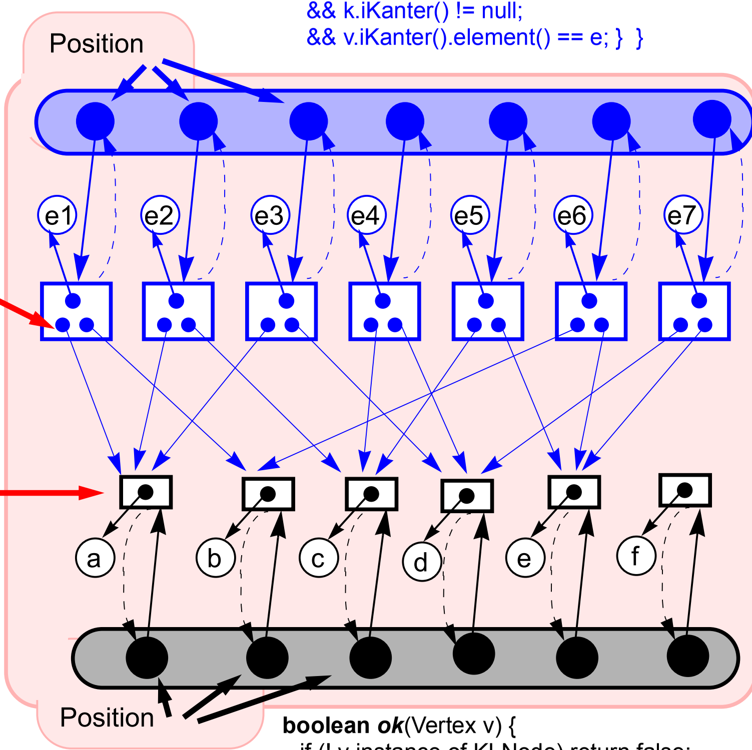

```
class KLEdge  implements Edge {
  protected Object elem; Container s;
  protected Position  kp;
  protected Vertex[] ee=new Vertex[2];
- - - - - - - - - - - - - - - - - - - -
  public KLEdge  (Vertex a, Vertex b,
                  Object o, Container n)  {
     elem=o; ee[0]=a; ee[1]=b; s=n; }
  public Vertex [] endV () { return ee; }
  public boolean  has(Vertex v) {
     return (ee[0]==v  | |  ee[1]==v) ; }
  public Position  iKanter () { return kp; }
  public void  setPos (Position p) { kp = p; }
 ... }
```

DataInvariant:

e . iKanter() . element() == e
n . iNoder() . element() == n

```
class KLNode  implements Vertex {
  protected Object elem; int deg=0;
  protected Container s;
  protected Position  np;
- - - - - - - - - - - - - - - - - - - -
  public KLNode (Object o,Container n)  {
     elem=o; s=n; }
  public Position iNoder  () {return np; }
  public void setPos  (Position p) { np = p; }
  public int degree  () { return deg; }
  public void inc () { deg = deg+1; }
  public void dec () { deg = deg–1; }
... }
```

public int **numEdges**() { kanter.size(); }
public int **numVertices**() { noder.size(); }
. . .




```
boolean ok(Edge e) {
  if (! e instance of KLEdge) return false;
  else {   KLEdge k = (KLEdge) e;
    return k.container() == this;
           && k.iKanter() != null;
           && v.iKanter().element() == e; } }
```

```
boolean ok(Vertex v) {
  if (! v instance of KLNode) return false;
  else {   KLNode n = (KLNode) n;
    return n.container() == this;
           && n.iNoder() != null;
           && n.iNoder().element() == n; } }
```

```
Edge insertEdge(Vertex v,u, Object o) {
  if (ok(v) && ok(u) && !areAdjacent(v,u))
  { KLEdge e = new KLEdge(v,u,o,this);
    e . setPos ( kanter . insertLast(e) );
    ((KLNode) v).inc(); ((KLNode) u).inc();
    return e;
  } else if (areAdjacent(v,u))  throw ...
  else throw new InvalidPosExc("");  }
```

```
Object removeEdge(Edge e) {                    O(1)
  if (ok(e)) {  KLEdge k = (KLEdge)e;
    Object ret= k.element();
    kanter.remove(k . iKanter());              !!!
    ( (KLNode) k.endV()[0] ).dec();
    ( (KLNode) k.endV()[1] ).dec();
    return ret; }
  else throw new InvalidPosExc(""); }
```

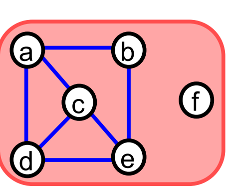
```
Vertex insertVertex(Object o) {
  KLNode v = new KLNode(o,this);
  v . setPos ( noder . insertLast(v) );
  return v; }
```

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Enumeration k= kanter.elements();
    while ( k.hasMoreElement() ) {             O(k)
      KLNode e= (KLNode)k.nextElement();
      if (e.has (v)) removeEdge(e);  }
    noder.remove(((KLNode)v).iNoder());  !!!
    return ret;
  } else throw new InvalidPosExc("");  }
```

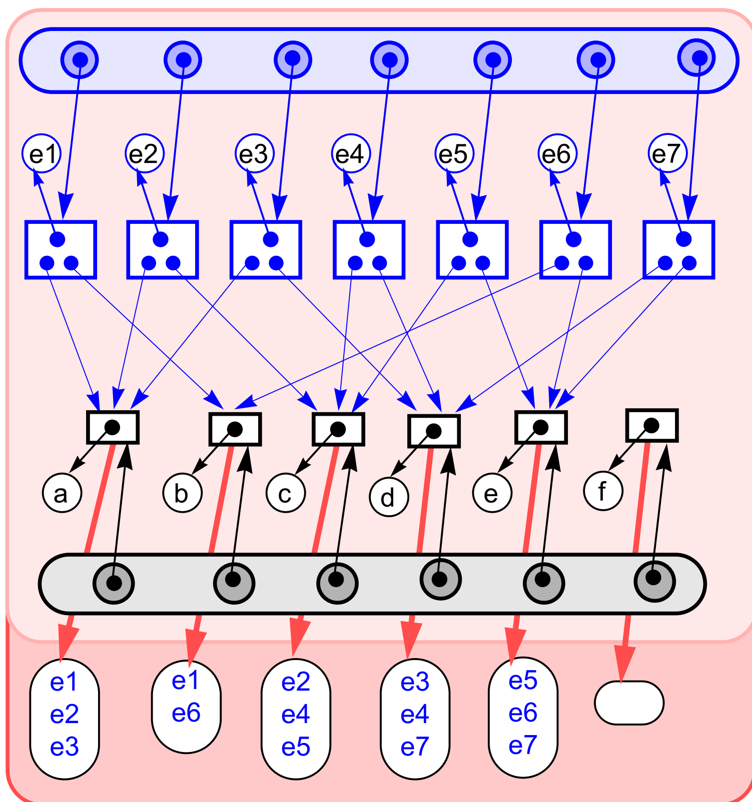

2. Implementasjon av Graph *med Nabo-Liste*

er som Kant-Liste men i tillegg

- hver node v har en samling Inc(v) med sine (incident) kanter
- (hver kant (u,v) holder en referanse til sin posisjon i Inc(v) og Inc(u))



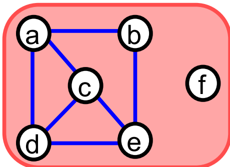
endV(e), opposite(v,e), degree(v)	$O(1)$	$O(1)$
insertV(o), insertE(v,u,o), removeE(e)	$O(1)$	$O(1)$
incidentE(v), adjacentV(v)	$O(k)$	$O(\deg v)$
removeV(v)	$O(k)$	$O(\deg v)$
areAdjacent(v,u)	$O(k)$	$O(\deg v\|u)$




```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Enumeration k= ((NLNode) v) . inc();
    while (k.hasMoreElements() ) {   O(deg v)
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc("");  }
```

- Inc(v) har ofte kun nabo-noder

3. Implementasjon av Graph *med Nabo-Matrise*



Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
0	1	2	3	4	5	no

i en **n xn** matrise **boolean A[n][n]**

- **A[i][j] == true** hviss G har en kant e=(i,j)
- **A[i][j] == false** hviss (i,j) ikke er en kant i G

	0	1	2	3	4	5
0		•	•	•		
1	•				•	
2	•			•	•	
3	•		•		•	
4		•	•	•		
5						

a b c d e f

plass forbruk $O(n^2)$

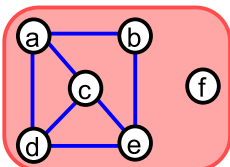
endV(e), opposite(v,e), degree(v)	$O(1)$	$O(1)$
insertE(v,u,o), removeE(e) ..	$O(1)$	$O(1)$
insertV(o)	$O(1)$	$O(n^2)$
incidentE(v), adjacentV(v)	$O(\deg v)$	$O(n)$
removeV(v)	$O(\deg v)$	$O(n^2)$
areAdjacent(v,u)	$O(\deg v\vert u)$	$O(1)$

```
boolean areAdjacent(Vertex v, Vertex u)
  { if ( ok(v) && ok(u) )
     return A [no(v)] [no(u)] ; }

void insertEdge(Vertex v,u) {
  if ( ok(v) && ok(u) )
     A [no(v)] [no(u)] = true ;
  } else if ( areAdjacent(v,u) ) throw ...
  else throw new InvalidPosExc(""); }
```

DataInvariant for ikke-rettet graf:
A [i] [k] == A [k] [i]

3. Implementasjon av Graph *med Nabo-Matrise*



Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
0	1	2	3	4	5	no

i en **n xn** matrise **A**

- **A[i][j] == e (true)** hviss G har en kant e=(i,j)
- **A[i][j] == null (false)** hviss (i,j) ikke er en kant i G

e1 e2 e3 e4 e5 e6 e7

	0	1	2	3	4	5
0		•	•	•		
1	•				•	
2	•			•	•	
3	•		•		•	
4		•	•	•		
5						

a b c d e f

plass forbruk $O(n^2)$

endV(e), opposite(v,e), degree(v)	$O(1)$	$O(1)$
insertE(v,u,o), removeE(e) ..	$O(1)$	$O(1)$
insertV(o)	$O(1)$	$O(n^2)$
incidentE(v), adjacentV(v)	$O(\deg v)$	$O(n)$
removeV(v)	$O(\deg v)$	$O(n^2)$
areAdjacent(v,u)	$O(\deg v\vert u)$	$O(1)$

```
Edge insertEdge(Vertex v,u, Object o) {
  if (ok(v) && ok(u) && !areAdjacent(v,u))
  { NMEdge e = new NMEdge(v,u,o,this);
     e . setPos ( kanter . insertLast (e) );
     ((KLNode) v).inc(); ((KLNode) u).inc();
     A[no(v)] [no(u)] = e ;
     return e;
  } else if (areAdjacent(v,u)) throw ...
  else throw new InvalidPosExc(""); }
```

*Utmerket for **stabile**, nesten **komplette** grafer (ikke for traversering)*

Implementasjoner av Graph

n : antall noder **k** : antall kanter

kompleksitet **operasjon**	**Kant-Liste**	**Nabo-Liste**	**Nabo-Matrise**
size, isEmpty	1	1	1
newContainer	1	1	1
elements, positions	n+k	n+k	n+k
replace, swap	1	1	1
numVertices(), numEdges()	1	1	1
vertices() / edges()	n / k	n / k	n / k
degree(v)	1	1	1
endVertices(e), opposite(v,e)	1	1	1
adjacentVertices(v), incidentEdges(v)	k	deg v	n
insertVertex(o)	1	1	n^2
removeVertex(v)	k	deg v	n^2
insertEdge(v,u,o)	1	1	1
removeEdge(e)	1	1	1
areAdjacent(v,u)	k	deg u|v	1

Oppsummering

- *Graf – definisjon, terminologi*
  ikke-rettet
  rettet
- ***Traversering** av Grafer*
  DFS – rekursiv algoritme
  – basis for andre algoritmer
  DFS tre, sammenhengende komponent, sti mellom to noder, ...
  BFS – iterativ algoritme (med Queue) : O(n+k)
  *– Ford-Bellman algoritme : O(n*k)*
  – basis for andre algoritmer
  BFS tre, sammenhengende komponent, sti mellom to noder, ...
  ***kortest** sti mellom to noder*
- *Graph ADT og dens implementasjoner*
  Kant-Liste
  Nabo-Liste
  Nabo-Matrise