Grafer

I. GRAF

definisjon
terminologi

II. GRAF TRAVERSERING

DFS
BFS

III. GRAF ADT OG IMPLEMENTASJON

Kant-Liste og Nabo-Liste
Nabo-Matrise

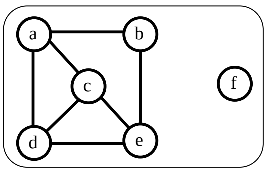
IV. RETTEDE GRAFER (DIGRAPHS)

V. VEKTEDE GRAFER

Kap. 9 (kursorisk 9.4.5)
Kap.10 (untatt 10.1.5, 10.2.2–10.2.4, 10.3)

En graf G

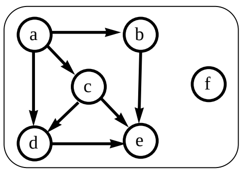
er gitt ved to mengder (E,V)
V av noder
E av kanter
der en kant e ε E er et (uordnet) par (u,v) av noder u, v ε V



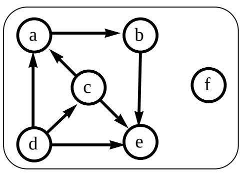
V = { a, b, c, d, e, f }

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 graf er rettet (diGraf) dersom
vi betrakter hver kant (u,v) som ordnet par 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) }

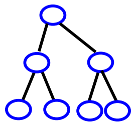
(En (ikke-rettet) graf kan sees som en rettet graf der for hvert par u,v ε V: hvis u → v ε E, så også v → u ε E.)

En mengde V med en binær relasjon E tilsvarer en (rettet) graf (V,E) – er relasjonen E symmetrisk, kan vi betrakte den som en ikke-rettet graf.

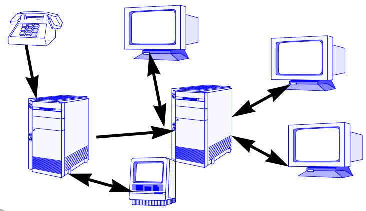
Anvendelser ...

I. BINÆRE RELASJONER 0 → 1 → 2 → 3 → 4 →

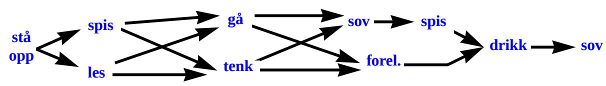
II. TRÆR



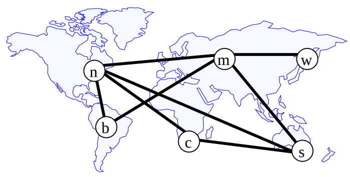
III. NETTVERK

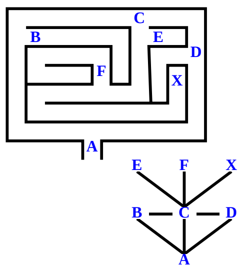


IV. PLANLEGGING



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

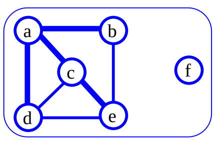
$$\sum_{v \in V} deg(v) = 2(\ \#\ \text{kanter}\ )$$

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}) ε E — acaca, bec, abedc, edce

enkel sti : ingen node forekommer 2 ganger

sykel : enkel sti men $n_1 = n_k$

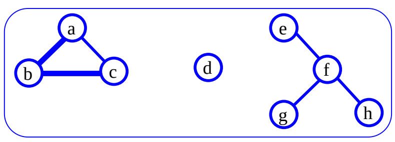


sammenhengende graf : det finnes en sti mellom alle par av noder

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)

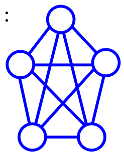
1. Telling av noder og kanter
a) Litt om sammenhenger...

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

• G er komplett hvis og bare hvis hver node har (***n***-1) naboer :

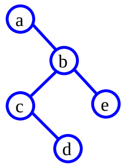
dvs. $k = 1/2 * \sum_{v \in V} deg(v) = 1/2 * n * (n-1)$

n =5
k =10



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

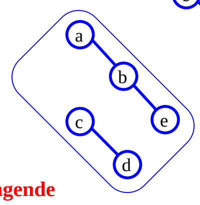
n =5
k =4



• **Hvis G er et tre så** $k = n-1$

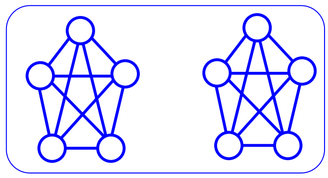
- et minimalt antall kanter som kan lage en sammenhengende graf av ***n*** noder

- fjerning av en vilkårlig kant, gjør grafen usammenhengende

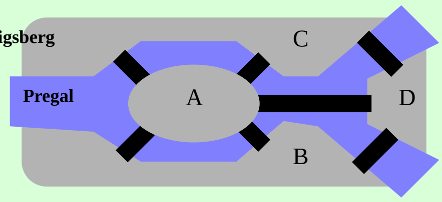


• **Hvis** $k < n-1$ **så er ikke G sammenhengende**
men ikke omvendt !!!

n =10
k =20 > 9



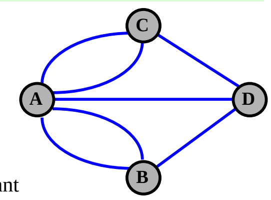
1.b) *Kants spasering og Eulers tur*



Kant :
Kan jeg krysse hver bro nøyaktig én gang og returnere til utgangspunktet ?

Euler (1736) :
Nei – og jeg kan bevise det v.hj.a. grafer !

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



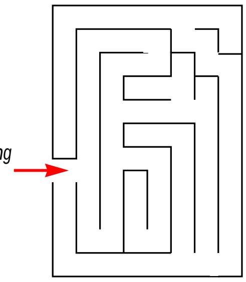
Eulers tur : en sti som traverserer hver kant nøyaktig én gang og returnerer til startnode

Eulers teorem : En graf G har en Eulers tur hvis og bare hvis graden til enhver node er et partall **O(n+k)**

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

... ...

Ariadnes (t)råd



Minos :

Fine, 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 kryss **u**, merk **u** '**visited**'.

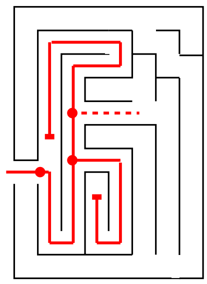
(**r**) Velg en vilkårlig vei – merk den og følg men **dra tråden** etter deg

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

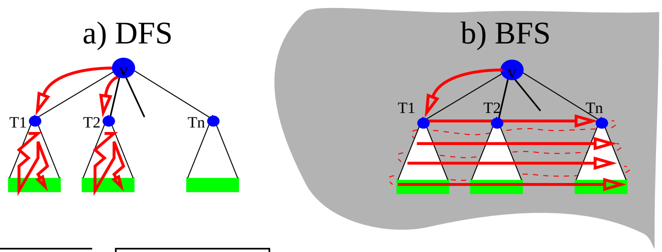
– gå da tilbake **langs tråden** til forrige krysset og fortsett å utforske umerkede veier derfra.

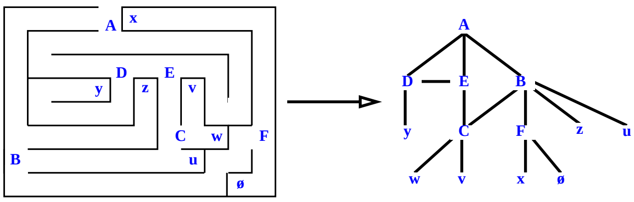
Du vil på den måten kunne utforske hele labirynten og returnere til inngangen

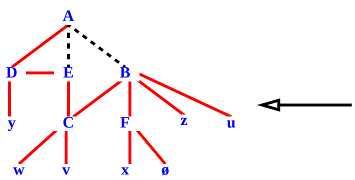
Chorus :

Smart Girl!

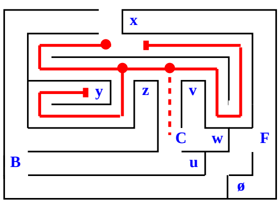
2.a) Graf traversering








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



Kanter merket med rød under DFS traversering gir et **utspennende tre for grafen**

– roten til treet kan velges vilkårlig !

DFS graf traversering

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) røde kanter gir et *utspennende tre*, kalt DFS tre, for sammenhengende komponenten til s

```
DFS(u)
  merk–u
  for hver kant e=(u,v)
     if ( ! merk-et–v )
        merk–e–rød
        DFS(v)
```

Begrunnelse :

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

- Da v er den første slik, 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)

Kjøretid av DFS:

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

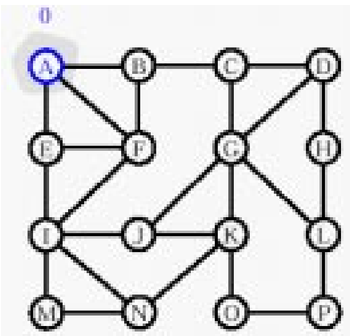
- gitt en kant, kan man aksessere dens ende-noder i $O(\mathbf{1})$
- merking av noder/kanter og sjekking om de er merket tar $O(\mathbf{1})$
- for hver node v, kan alle dens kanter akssesere 1 gang i $O(\mathbf{k(v)})$

DFS kan brukes for å lage $O(\mathbf{n+k})$ algoritmer for å :

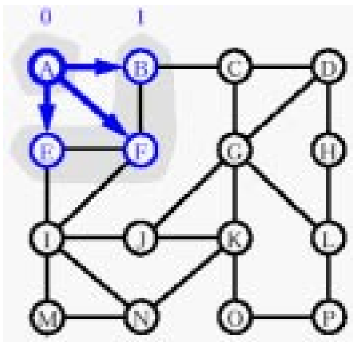
- sjekke om G er sammenhengende og finne sammenhengende komponenter
- finne utspennende tre for G
- sjekke om det finnes en sti mellom to noder;
- sjekke om G inneholder sykler

2.b) BFS graf traversering

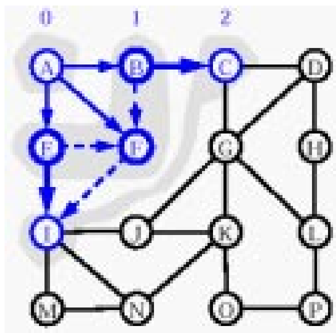
Hva er lengden av korteste vei fra A til P ?



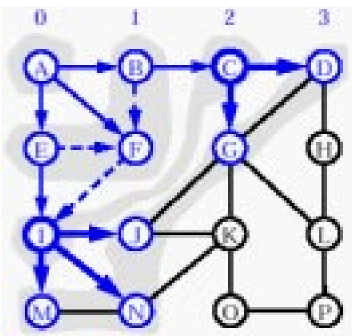
b)



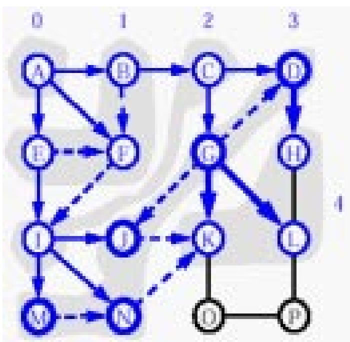
c)



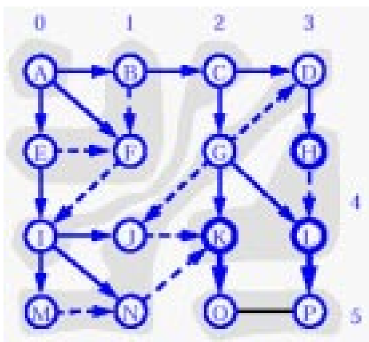
d)



e)



f)



2.b) BFS graf traversering

Tree DFS

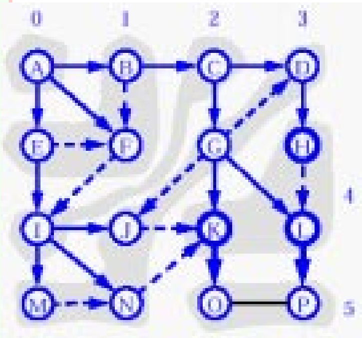
/*DFS(Tree T, Position v)
* **Stack S = new Stack()**
* S.push(v)
* while (!S.isEmpty())
* p= S.pop()
* for each p's child c
* S.push(c)
*/

Tree BFS

/*BFS(Tree T, Position v)
* **Queue S = new Queue()**
* S.enqueue(v)
* 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 v)
   Queue S = new Queue()
   mark(v, 0) – merker med nivå
   S.enqueue(v)
   while (!S.isEmpty())
     p= S.dequeue()
     for each kant e=(p,c)
       if (! marked(c))
          mark(c, p.mark+1)
          mark(e, rød)
          S.enqueue(c)
```



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

- sjekke om G er sammenhengende
- finne sammenhengende kompo-nenter 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*

Graf ADT

// Noder og kanter er Position
public interface Graph extends PosSamling
{ int **numVertices();**
int **numEdges();**
~~Enumeration **vertices();**~~
~~Enumeration **edges();**~~
int **degree(**Position v**);**

int **size();** boolean **isEmpty();** ~~Enumeration **elements();**~~
~~Enumeration **positions();**~~ void **swap(**Position p, Position q**);** Object **replace(**Position p, Object o**);**

/** @param v en node-Position
* @return nabo noder */
PosSequence **adjacentV(**Position v**);** // ~~Enumeration~~

/** @param v en node-Position
* @return kanter til/fra noden */
PosSequence **incidentE(**Position v**);** // ~~Enumeration~~

/** @param v en node-Position
* @param e en kant-Position
* @return noden i motsatt enden av e */
Position **opposite(**Position v, Position e**);**

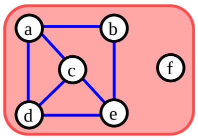
/** @param e en kant-Position
* @return array med to noder i endene av e */
Position[] **endV(**Position e**);**

/** @param v,u to node-Position
* @return true hviss det finnes en kant (v,u) */
boolean **areAdjacent(**Position v, Position u**);**

// Oppdatering:
/** sett inn og returner en ny kant (v,u) med Object o; v, u er i node-mengde fra før */
Position **insertE(**Position v, Position u, Object o**);**

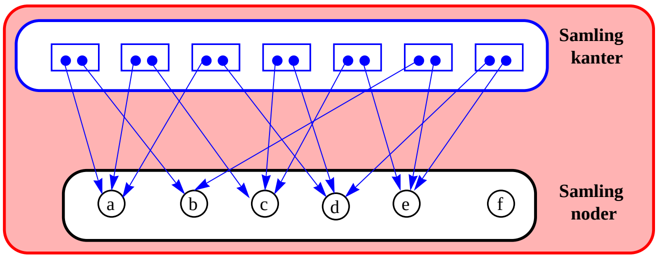
/** sett inn og returner en ny isolert node med Object o */
Position **insertV(**Object o**);**

/** returner objektet lagret i node v; fjern v samt alle kanter til denne*/
Object **removeV(**Position v**);**
Object **removeE(**Position e**);**
}

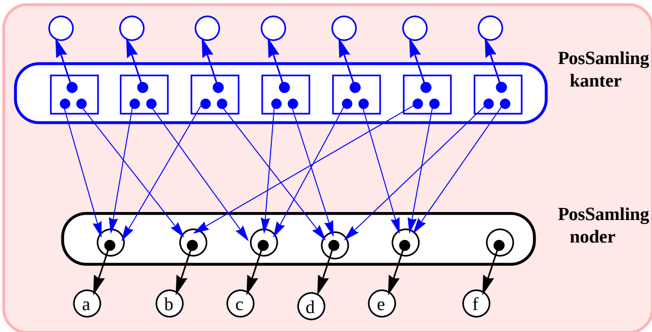


Implementasjon av Graph
1) Kant-Liste

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



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



1) Kant-Liste DS

```
class KLNode implements Position {
  protected Object elem; protected Samling s;
  protected int degree=0;
  public KLNode(Object o, Samling ns)  { elem = o;  s= ns;}
  public int degree()
      { return degree; }
  public void incDegree()
      { degree++; }
  ... }
```

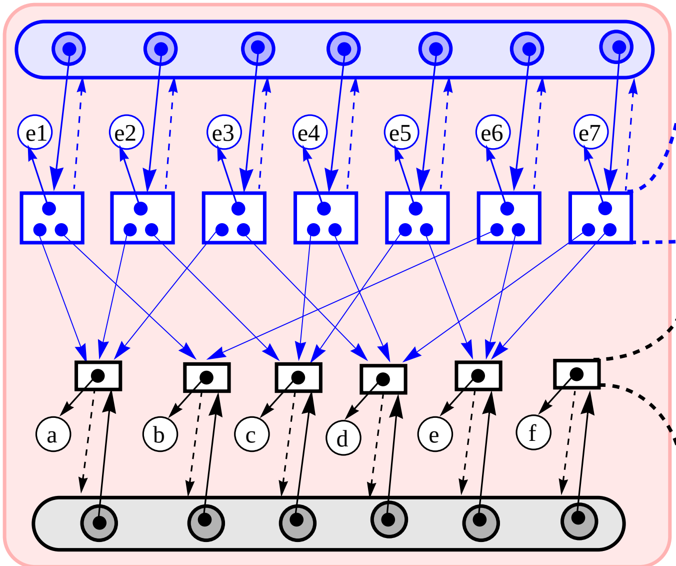
```
class KLEdge implements Position {
  protected Object elem; protected Samling s;
  protected KLNode[] ends=new KLNode[2];
  public KLEdge(Object o,KLNode a,KLNode b,Samling n)
      { elem=o; ends[0]=a; ends[1]=b; s=n; }
  public Position[] endV() { return ends; }
  public boolean has(Position v)
      { return (ends[0]==v || ends[1]==v) ; }
  ... }
```

```
class GraphKL implements Graph {
  private PosSamling          noder= new ???,
                              kanter= new ???;
  private int noN= 0, noE= 0;
  public Position insertV(Object o)
     {   KLNode n= new KLNode(o, noder);
         ((???)noder).insert(n); noN++; return n; }
  public Position insertE(Position v, Position u, Object o)
     {   KLEdge e= new KLEdge(o, (KLNode)v, (KLNode)u, kanter);
         ((KLNode)v).incDegree(); ((KLNode)u).incDegree();
         ((???)kanter).insert(e); noE++; return e; }
  public Object removeV(Position v)
     {   gå gjennom kanter og fjern hver kant e slik at e.has(v)
         ((???)noder).remove(v); noV--; }
  public PosSequence adjacentV(Position v)
     {   gå gjennom kanter og ta med hver kat slik at e.has(v)}
  public int degree(Position v)  {return ((KLNode)v).degree();}
... }
```

PosSequence
RankedSequence – numerert
Dictionary – signifikant nøkkel

public class GraphKL implements Graph {

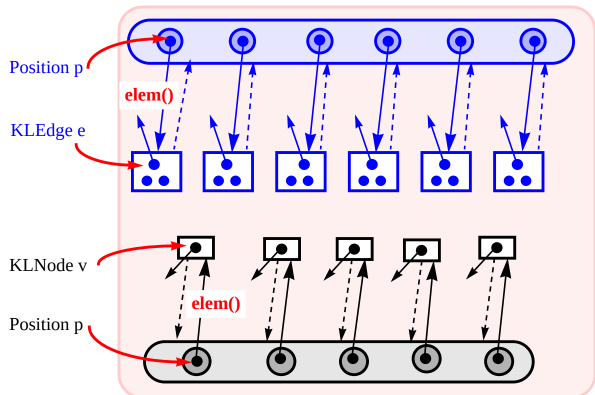
private PosSequence kanter = new ??? ;
private int noE = 0;
public int numEdges() { return noE; }




```
class KLEdge implements Position {
  protected Object elem;
  protected Samling s;
  protected KLNode[]
      ends=new KLNode[2];
  public KLEdge
   (Object o,KLNode a,KLNode b,Samling n)
      { elem=o; ends[0]=a; ends[1]=b; s=n; }
  public Position[] endV() { return ends; }
  public boolean has(Position v)
      { return (ends[0]==v || ends[1]==v) ; }
  ... }
```

```
class KLNode implements Position {
  protected Object elem;
  protected Samling s;
  protected int degree=0;
  public KLNode(Object o, Samling ns)
      { elem = o;  s= ns;}
  public int degree()
      { return degree; }
  public void incDegree()
      { degree++; }
  ... }
```

private PosSequence noder = new ??? ;
private int noV = 0;
public int numVertices() { return noV; }




```
private boolean erNode(Position v) {
   if (v==null) return false;
   else if (v instanceof KLNode)
      return v.samling() == noder;
   else return false; }
// p er en Position fra sekvens noder
private KLNode nodeAt(Position p) {
  return (KLNode)p.elem(); }
// v er en KLNode
private Position niSeq(Position v) {
  Position p = noder . first();
  boolean fant = false;
  while (!fant) {
    if (p.elem() == v) fant=true;
    else p = noder . after(p); }
  catch(LastPos ex) { p= null; }
  return p; }                      O(n)/O(1)
```

```
private boolean erKant(Position e) {
   if (e==null) return false;
   else if (e instanceof KLEdge)
      return e.samling() == kanter;
   else return false; }
// p er en Position fra sekvens kanter
private KLEdge edgeAt(Position p) {
  return (KLEdge)p.elem(); }
// e er en KLEdge
private Position eiSeq(Position e) {
  Position p = kanter . first();
  boolean fant = false;
  while (!fant) {
    if (p.elem() == e) fant=true;
    else p = kanter . after(p); }
  catch(LastPos ex) { p= null; }
  return p; }                      O(k)/O(1)
```

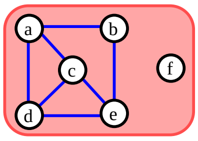
...

```
public Position insertV(Object o) {                                          O(1)
   noV++;  return nodeAt( noder . insertFirst(new KLNode(o, noder)) ) ;  }
public Position insertE(Position v, Position u, Object o)
                                    throws InvalidPos, EdgeExists {
   if (!erNode(v) || !erNode(u)) throw new InvalidPos("noder ikke i grafen");
   if (areAdjacent(v, u)) throw new EdgeExists("noder har en kant");
   else { KLEdge e= new KLEdge(o, (KLNode)v, (KLNode)u, kanter);
          ((KLNode)v).incDegree(); ((KLNode)u).incDegree();
          kanter . insertFirst(e);  noE++;
          return e;  }
public Object removeE(Position e)  {                                         O(1)
   if (erKant(e)) {
      Object r = e.elem();
      noE--;  kanter.remove(eiSeq(e));  return r; }                       !! O(k)
   else return null; }
public Object removeV(Position v)  {                                         O(k)
   if (!erNode(v)) return null;
   else { Object r = v.elem();
          Position p = kanter . first() ;
          while (true) { KLEdge e = edgeAt(p);                              O(k)
             if (e.has(v)) { ((KLNode) opposite(v,e)) . decDegree();
                kanter.remove(p); }
             else p = kanter.after(p) ;   }
          catch(LastPos e) { noder . remove(niSeq(v)); noV--; }           !! O(n)
          return r;  } }
public Position opposite(Position v, Position e)  {                          O(1)
   if (!erNode(v) || !erKant(e))  throw new InvalidPos("ikke i grafen");
   KLEdge ee = (KLEdge) e ; Position r = null;
   if (e.endV[0]==v) r = e.endV[1];
   else if (e.endV[1]==v) r = e.endV[0];
   return r; }
```



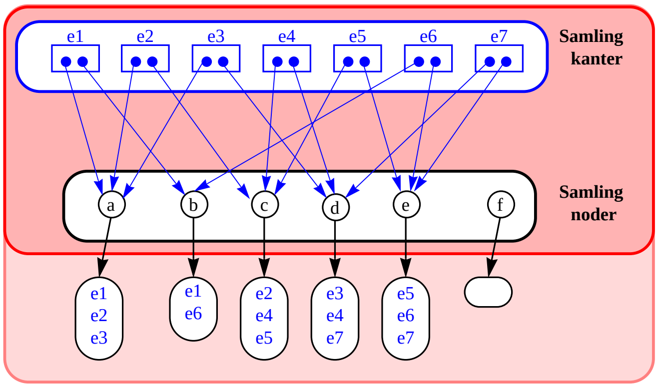
```
public PosSequence incidentE(Position v)  {
   if (erNode(v)) {
      PosSequence S = new ??? ;
      Position p = kanter . first() ;
      while (true) { KLEdge e = edgeAt(p);                                  O(k)
         if (e.has(v)) S.insertLast(e);
         p =  kanter . after(p) ;   }
      catch(LastPos ex) { }
      return S; }
   else throw new InvalidPos("ikke i grafen");   }
public PosSequence adjacentV(Position v)  {
   if (erNode(v)) {
      PosSequence S = new ??? ;
      Position p = kanter . first() ;
      while (true) { KLEdge e = edgeAt(p);                                  O(k)
         if (e.has(v)) S.insertLast(opposite(v,e));
         p = kanter . after(p) ;   }
      catch(LastPos ex) { }
      return S;  }
   else throw new InvalidPos("ikke i grafen"); }
public boolean areAdjacent(Position v, Position u) {
   if (erNode(v) && erNode(u)) {
      boolean are = false;
      Position p = kanter.first() ;
      while (!are) { KLEdge e = edgeAt(p);                                  O(k)
         if (e.has(v) && e.has(u)) { are= true; }
         else p = kanter.after(p) ;  }
      catch(LastPos ex) { } return are;  }
   else throw new InvalidPos("ikke i grafen"); }
public Position[] endV(Position e)  {                                       O(1)
   if (erKant(e)) return ((KLEdge)e).endV();
   else throw new InvalidPos(""); }
public int degree(Position v)  {                                            O(1)
   if (erNode(v)) return ((KLNode)v).degree(); else throw new InvalidPos(""); }
```

Implementasjon av Graph
2) Nabo-Liste DS



er som Kant-Liste men i tillegg

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

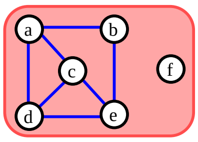


endV(e), opposite(v,e), degree(v)	*O*(**1**)	**O(1)**
insertV(o), insertE(o,v,u), removeE(e)	*O*(**1**)	**O(1)**
incidentE(v), adjacent(v)	*O*(**deg(v)**)	**O(k)**
areAdjacent(v,u)	*O*(**min{deg(v),deg(u)}**)	**O(k)**
removeV(v)	*O*(**deg(v)**)	**O(k)**

plass forbruk *O*(**k+n**)

I(v) er typisk implementert som Sequence
(ofte inneholder den bare noder, der det ikke trenges annen kant-informasjon)

Implementasjon av Graph
3) Nabo-Matrise DS

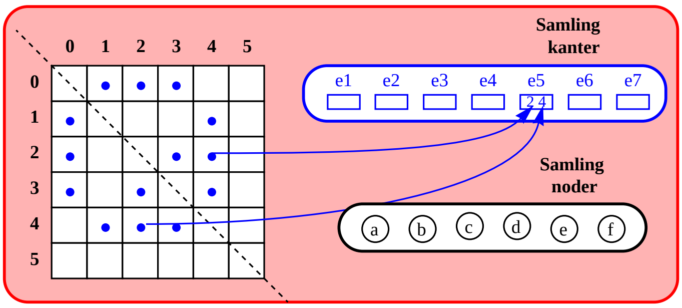


Enumerer alle noder fra 0...n-1:

a	**b**	**c**	**d**	**e**	**f**
|	|	|	|	|	|
0	**1**	**2**	**3**	**4**	**5**

i en **n x n** matrise **A**

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



endV(e), opposite(v,e), degree(v)	*O*(**1**)	**O(1)**
insertE(o,v,u), removeE(e)	*O*(**1**)	**O(1)**
removeE(v,u)	*O*(**1**)	
incidentE(v), adjacent(v)	*O*(**n**)	**O(deg(v))**
areAdjacent(v,u)	*O*(**1**)	**O(deg(v,u))**
insertV(o), removeV(v)	*O*($\mathbf{n^2}$)	**O(deg(v))**

plass forbruk *O*($\mathbf{n^2}$)

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