Grafer

I. GRAF

definisjon / terminologi

noen eksempler på graf-problemer

II. NOEN ENKLE GRAFALGORITMER

III. GRAF TRAVERSERING

DFS og BFS

IV. GRAF ADT OG IMPLEMENTASJON

Kant-Liste og Nabo-Liste

Nabo-Matrise

V. RETTEDE GRAFER

topologisk sorterting, transitiv tillukking

VI. VEKTEDE GRAFER

kortest sti, minste utspennede tre

Kap. 12 (kursorisk 12.4.4, 12.7.2)

IV. Graf ADT *(ikke-rettet utsnitt)*

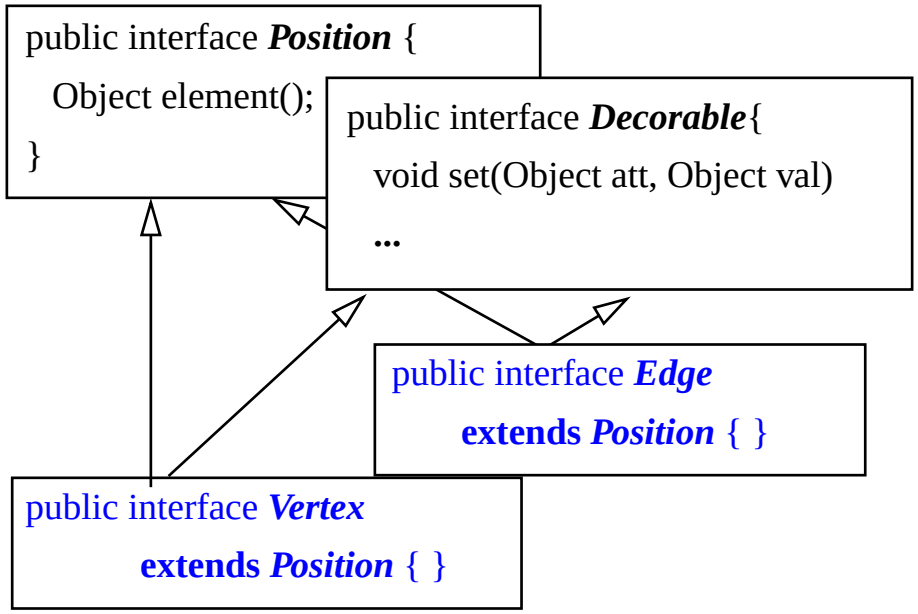
package jdsl.graph.api;

```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

IV. Graf ADT (*ikke-rettet utsnitt*)

package jdsl.graph.api;

```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

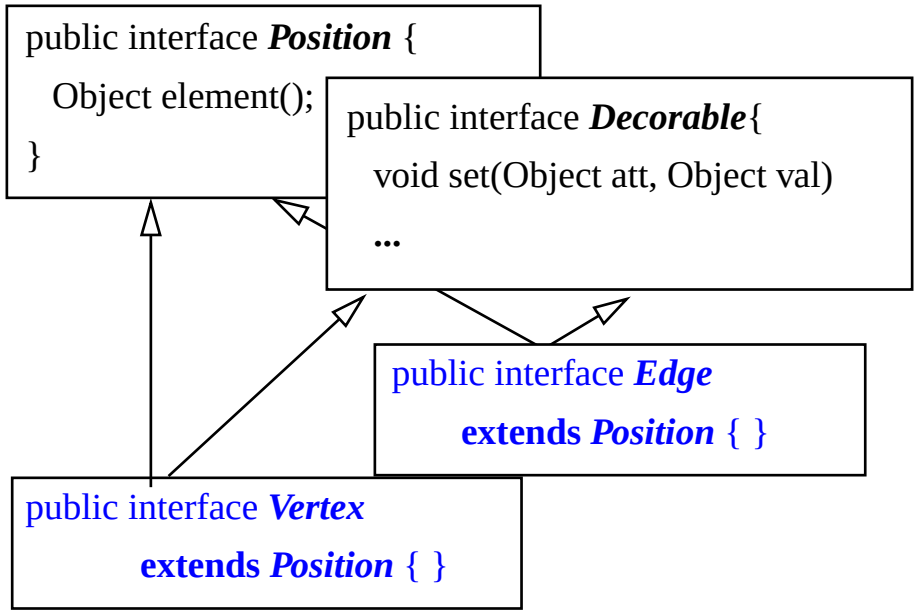


IV. Graf ADT *(ikke-rettet utsnitt)*

package jdsl.graph.api;

```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

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

   Object removeEdge(Edge e) ;
/** fjern noden v og ALLE kanter med v */
   Object removeVertex(Vertex v) ; !!!

}
```



IV. Graf ADT

(*kan ha **både** rettede og ikke-rettede kanter*)

package jdsl.graph.api;

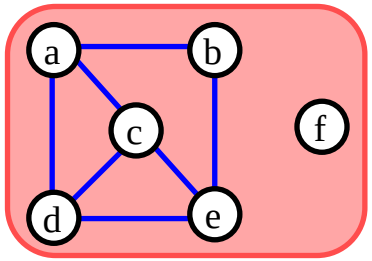
```
public interface InspectableGraph extends PositionalContainer {

  int numVertices();  int numEdges();
  VertexIterator vertices();
  EdgeIterator edges();
/** # rettede og ikke-rettede nabokanter */
  int degree(Vertex v);
  Vertex[] endVertices(Edge e)
  Vertex opposite(Vertex v, Edge e);
/** nabonoder langs alle kanter
    inn–, utgående samt ikke-rettede */
  VertexIterator adjacentVertices(Vertex v)
/** rettede og ikke-rettede nabokanter */
  EdgeIterator incidentEdges(Vertex v)
}
```

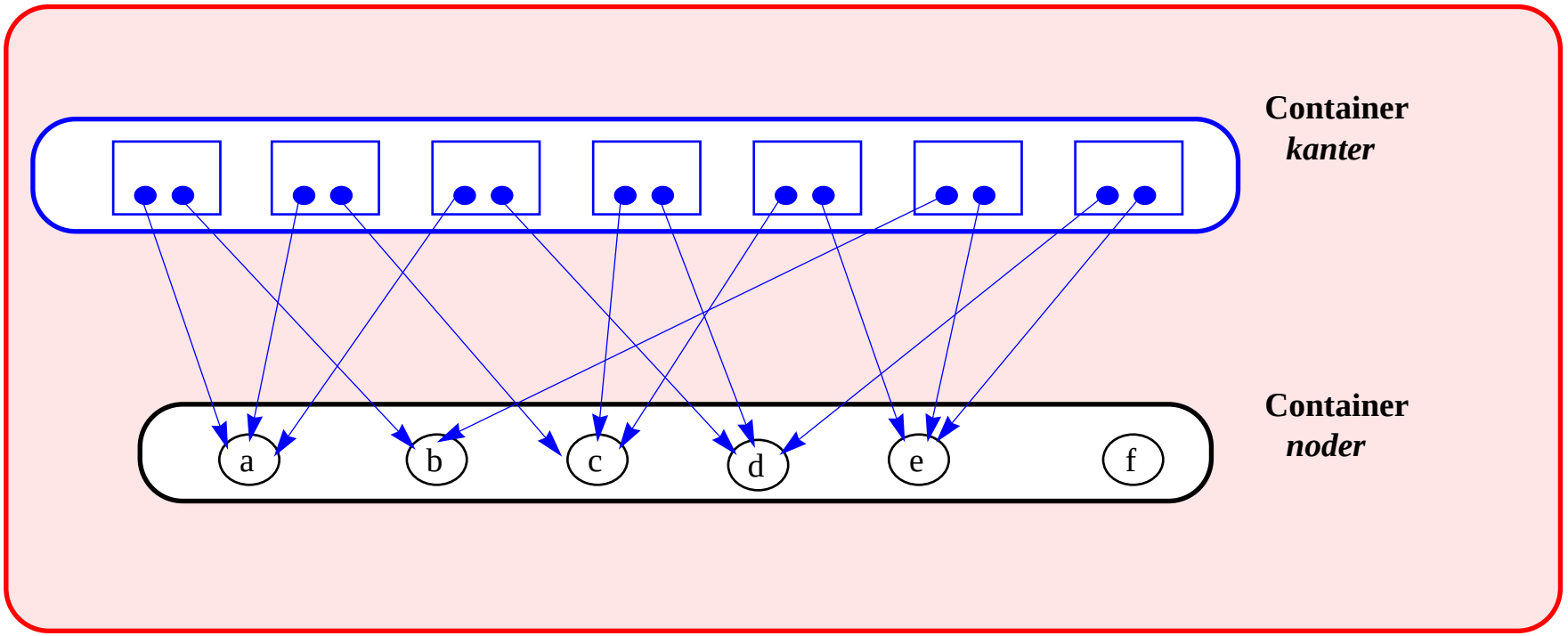
```
  EdgeIterator unDirectedEdges();
  EdgeIterator directedEdges();
  int outDegree(Vertex v);
  int inDegree(Vertex v);
  Vertex origin(Edge e)
  Vertex destination(Edge e)
  boolean isDirected(Edge e)
  VertedIterator outAdjacentVertices(Vertex v)
  VertedIterator inAdjacentVertices(Vertex v)
  EdgeIterator outIncidentEdges(Vertex v)
  EdgeIterator inIncidentEdges(Vertex v)
```

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

  Object removeEdge(Edge e)
  /** fjern noden v og ALLE kanter med v */
  Object removeVertex(Vertex v)

}
```

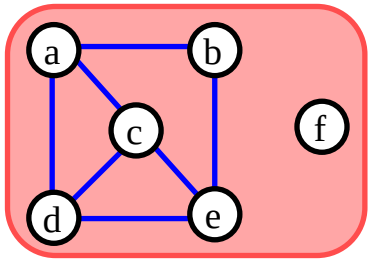
```
  void makeUndirected(Edge e)
  Edge insertDirectedEdge( Vertex u,
                           Vertex v, Object o);
  void reverseDirection(Edge e)
  /** gir retning til en ikke-rettet kant */
  void setDirectionTo/From(Edge e, Vertex v)
```

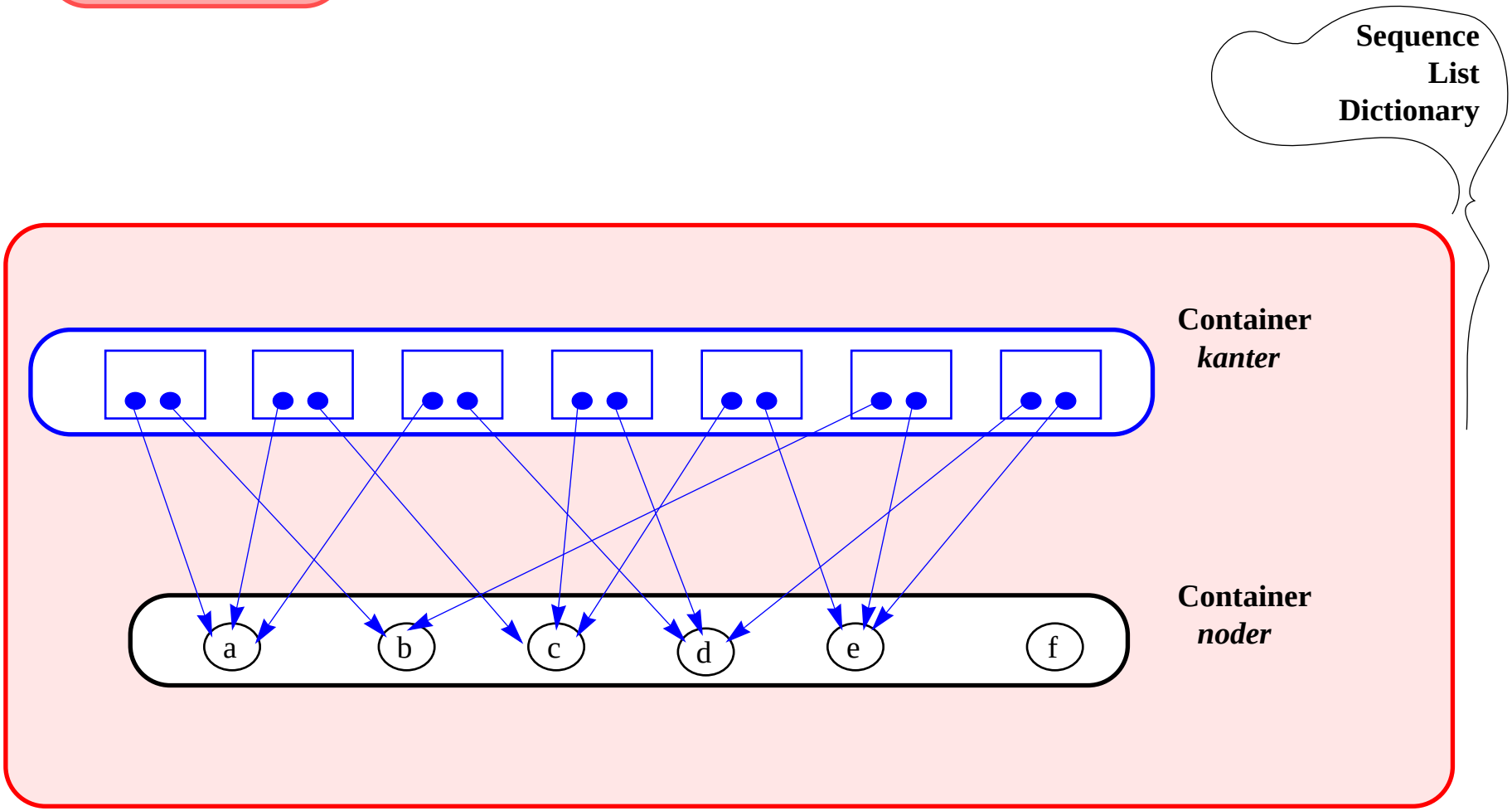
1. Implementasjon av Graph *med Kant-Liste*



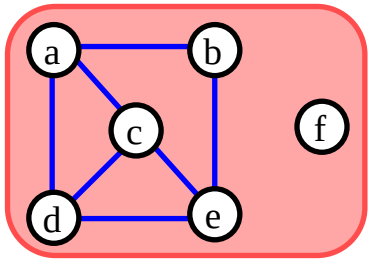


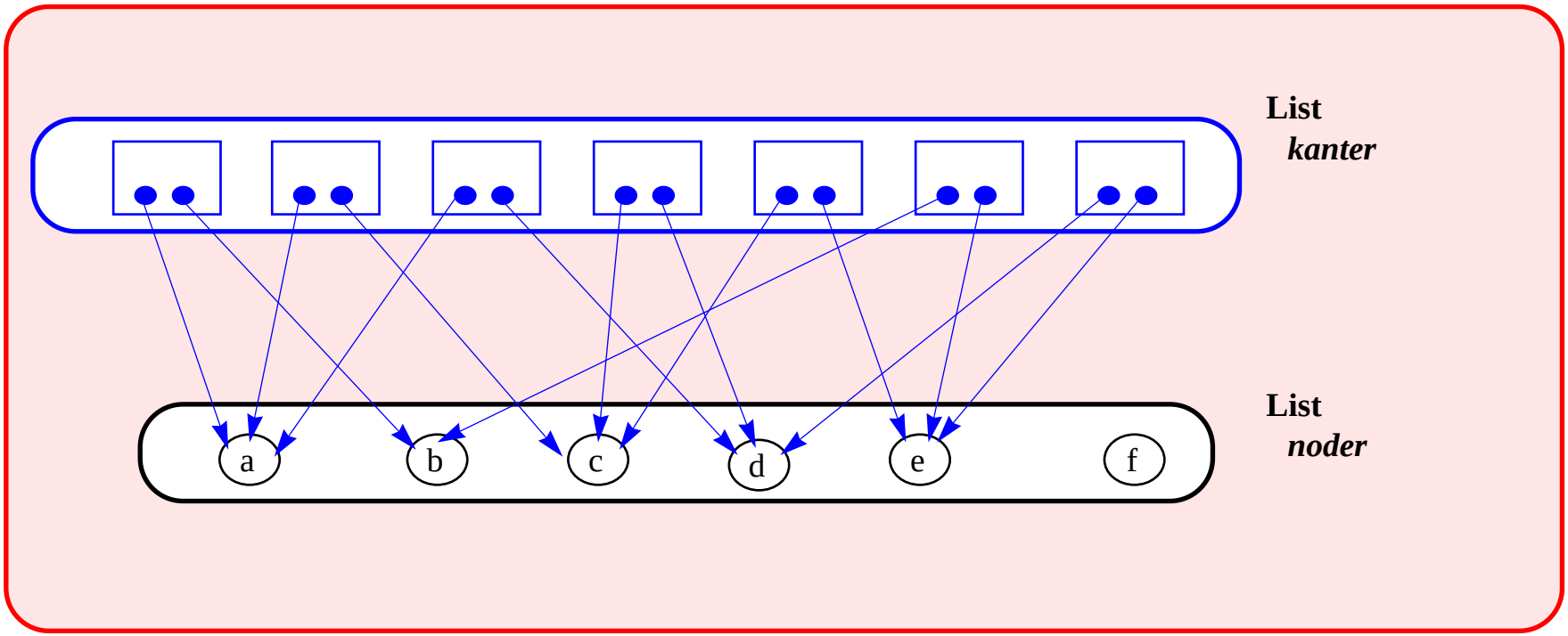
1. Implementasjon av Graph *med Kant-Liste*



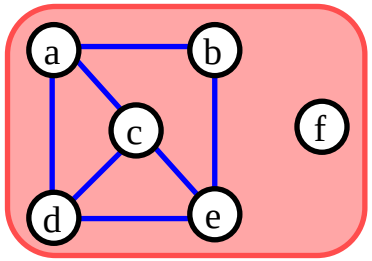


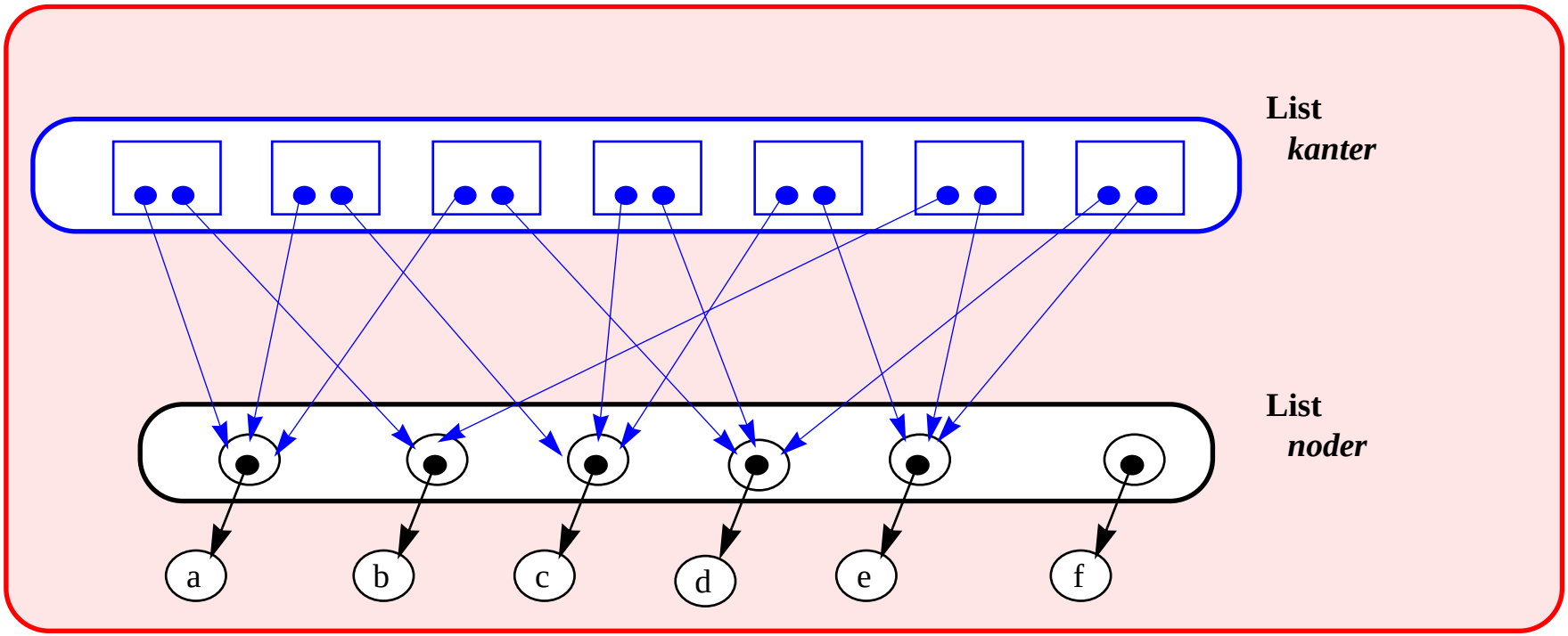
1. Implementasjon av Graph *med Kant-Liste*



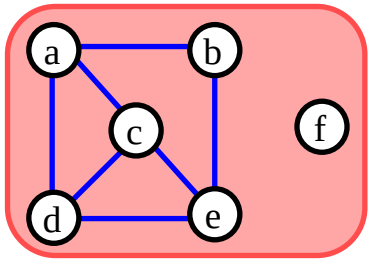


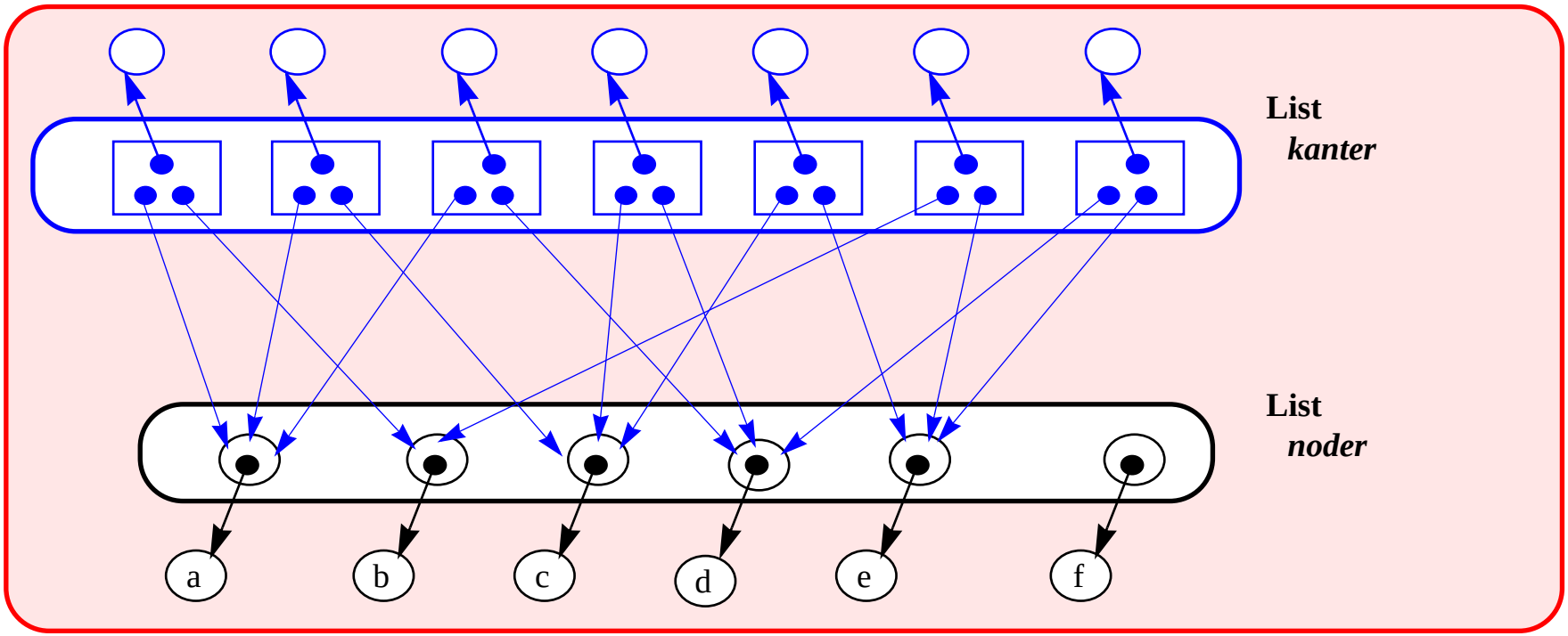
1. Implementasjon av Graph *med Kant-Liste*



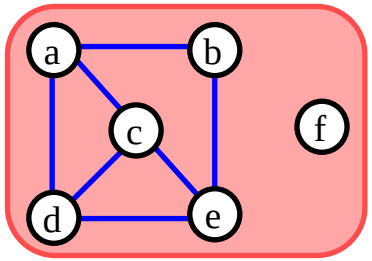


1. Implementasjon av Graph *med Kant-Liste*

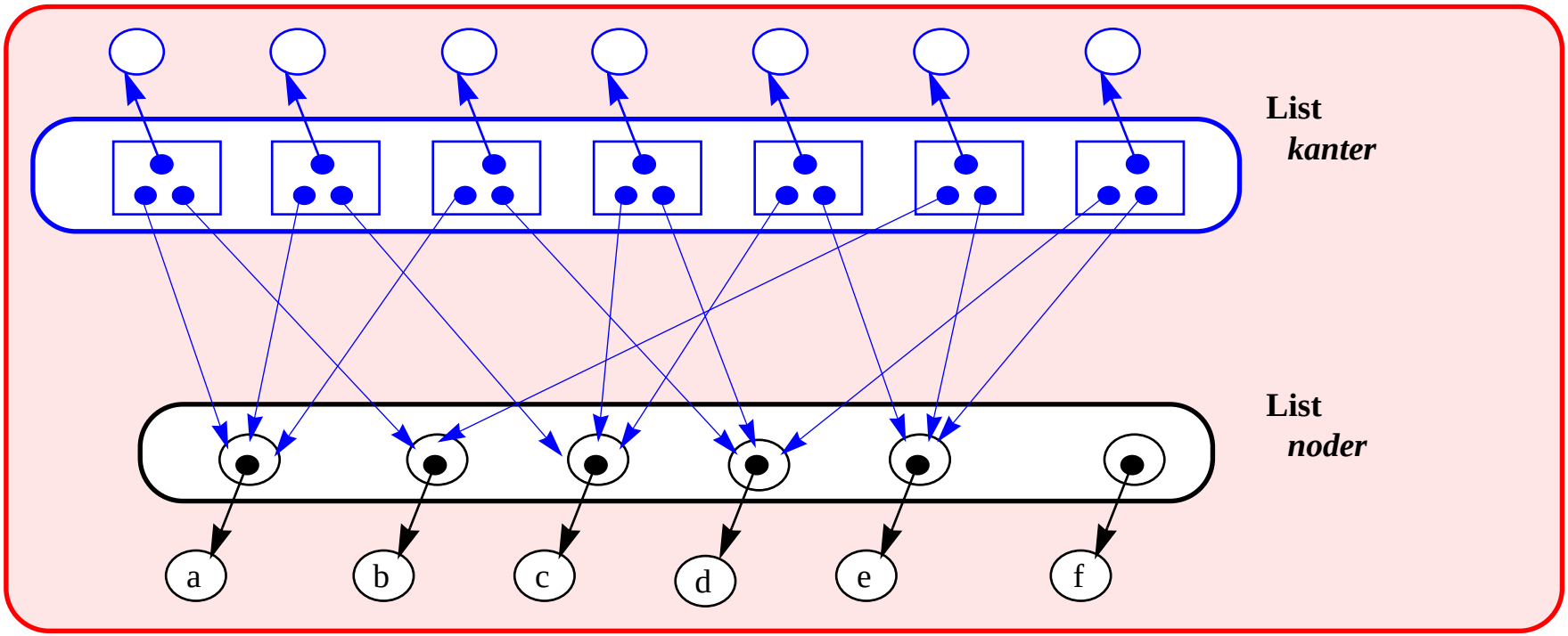




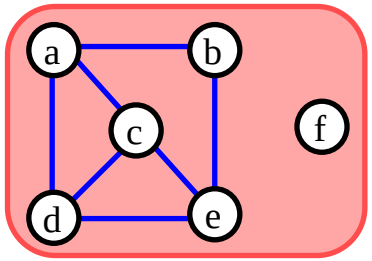
1. Implementasjon av Graph *med Kant-Liste*



endV(e), opposite(v,e),
degree(v) .....
insertV(o), insertE(v,u,o),
removeE(e) .....
incidentE(v), adjacentV(v) ..
removeV(v) .....
areAdjacent(v,u) .....



1. Implementasjon av Graph *med Kant-Liste*



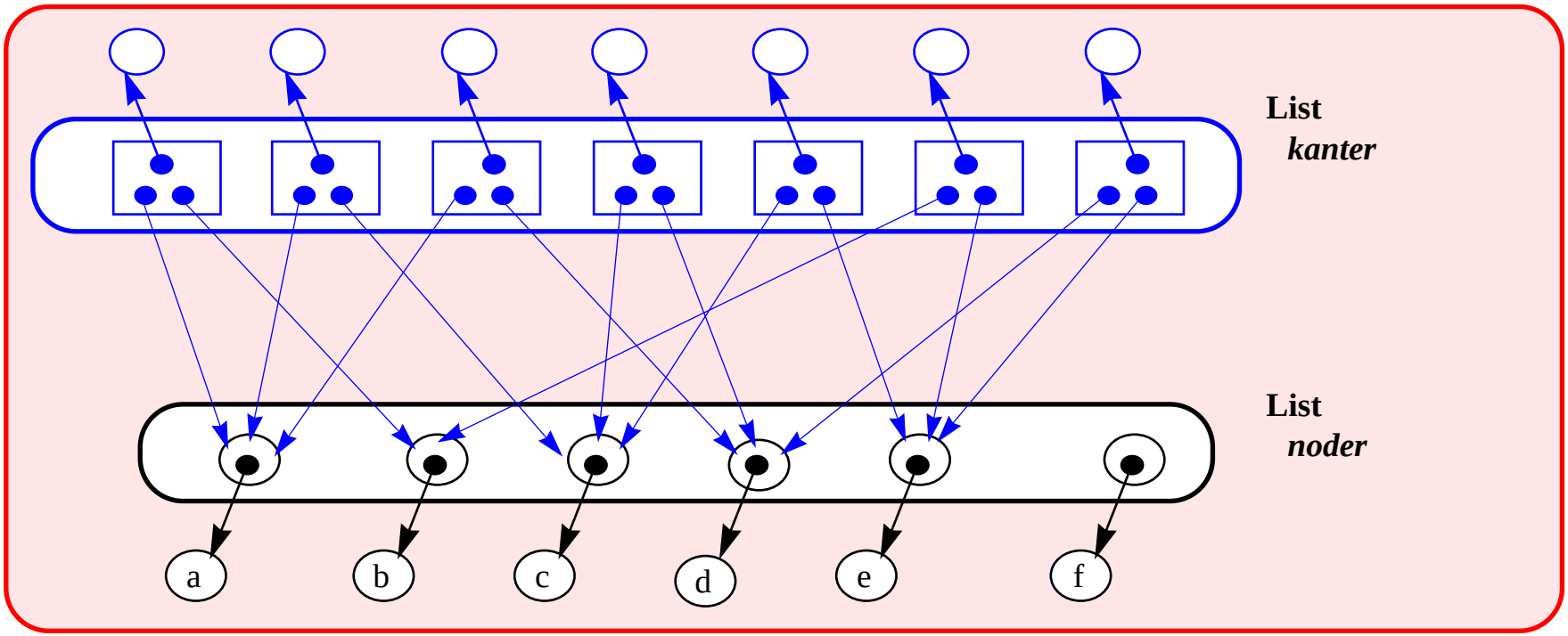
endV(e), opposite(v,e),
degree(v) ..... $O(1)$

insertV(o), insertE(v,u,o),
removeE(e) ..... $O(1)$

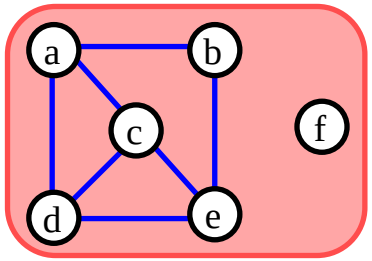
incidentE(v), adjacentV(v) ..

removeV(v) .....

areAdjacent(v,u) .....



1. Implementasjon av Graph *med Kant-Liste*



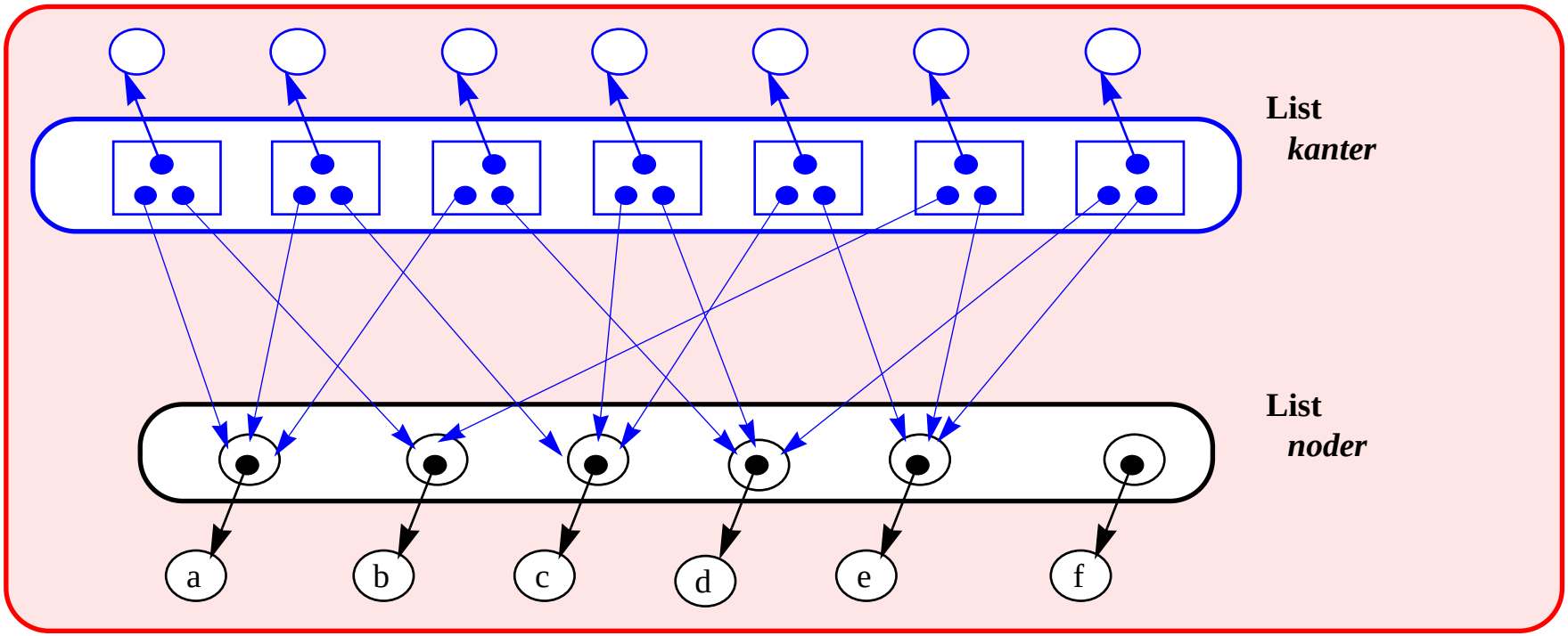
endV(e), opposite(v,e),
degree(v) ..... $O(1)$

insertV(o), insertE(v,u,o),
removeE(e) ..... $O(1)$

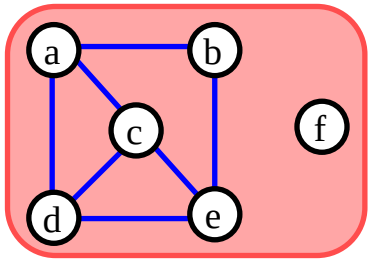
incidentE(v), adjacentV(v) .. $O(k)$

removeV(v) ..... $O(k)$

areAdjacent(v,u) ..... $O(k)$



1. Implementasjon av Graph *med Kant-Liste*



endV(e), opposite(v,e),
degree(v) ..... O**(1)**

insertV(o), insertE(v,u,o),
removeE(e) ..... O**(1)**

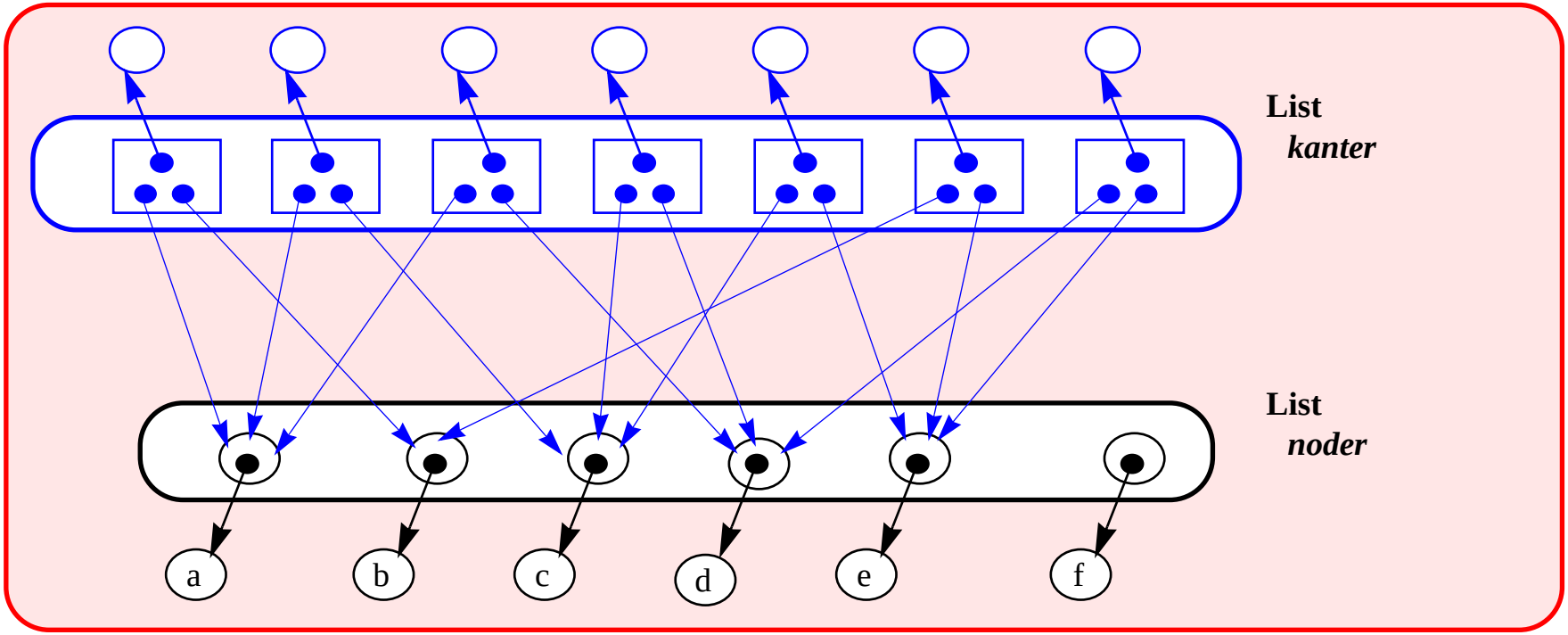
incidentE(v), adjacentV(v) .. O**(k)**

removeV(v) ..... O**(k)**

areAdjacent(v,u) ..... O**(k)**

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

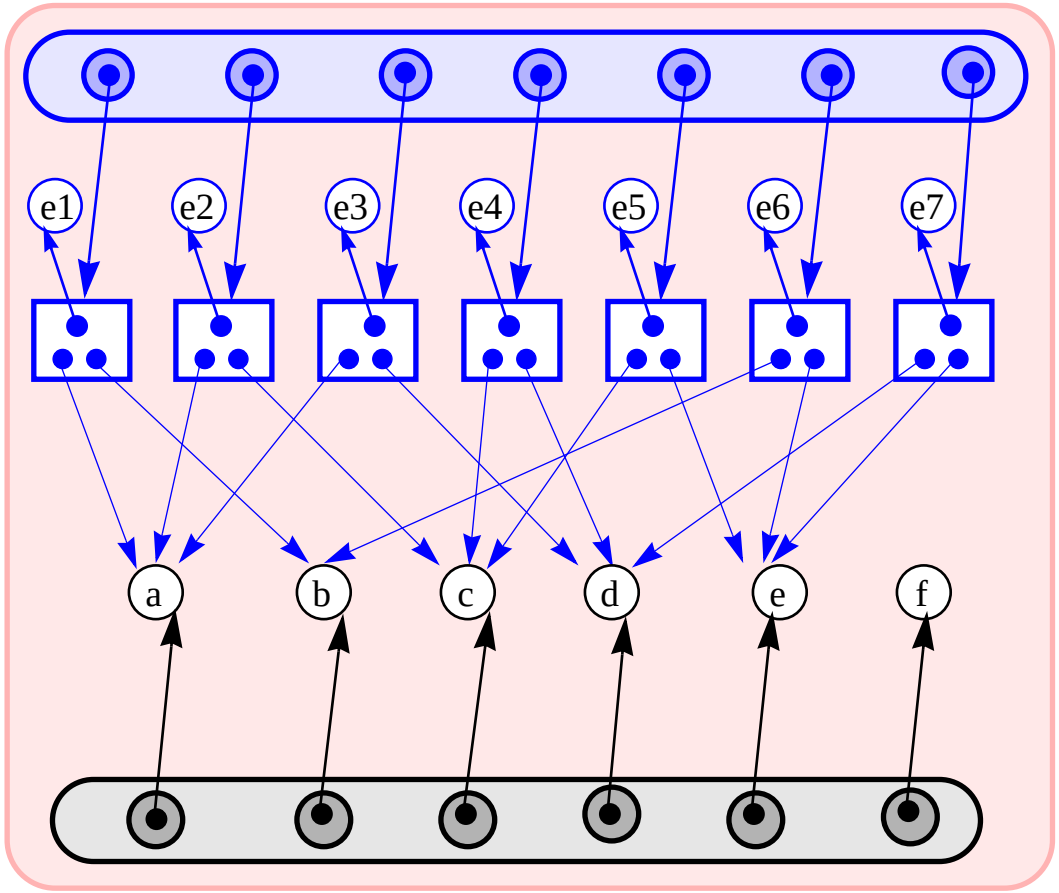
Sequence
List
Dictionary



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

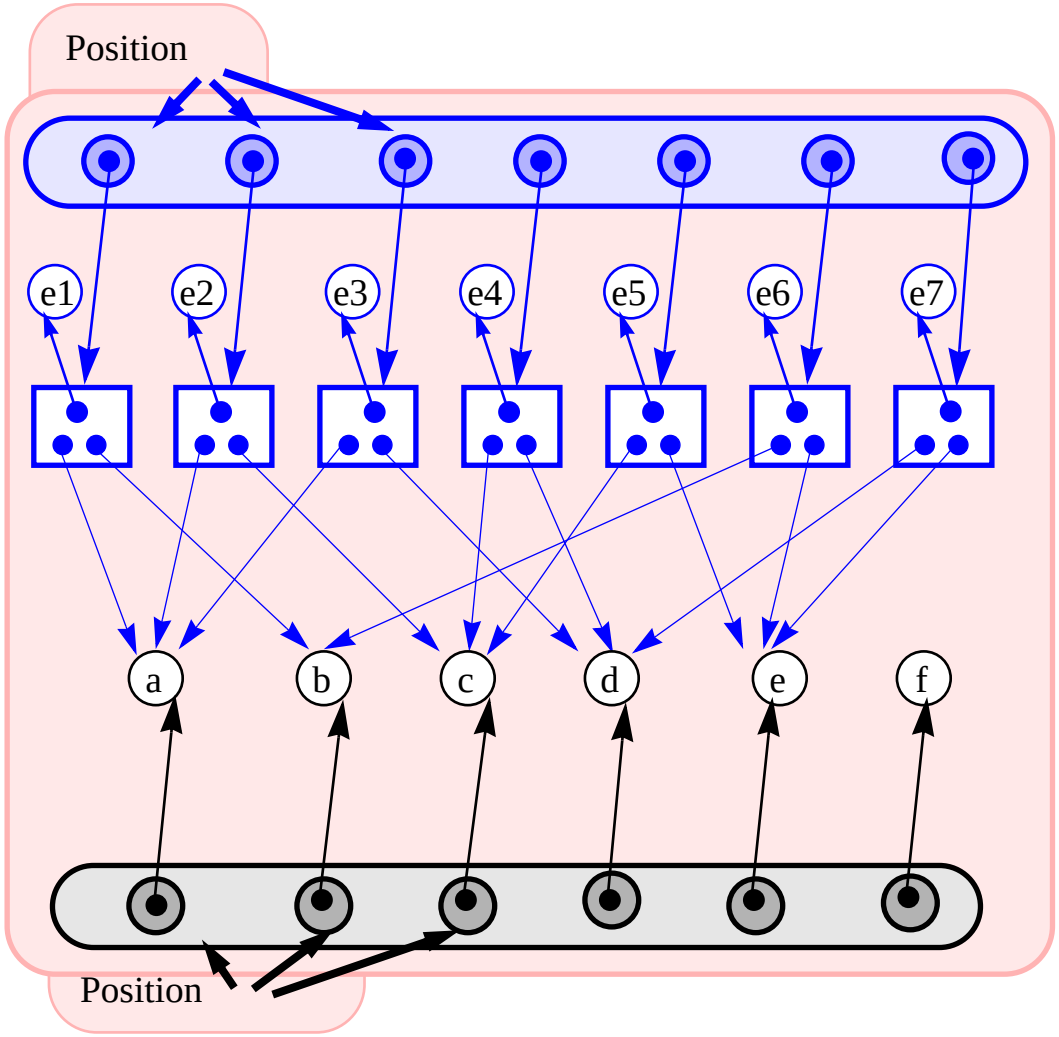
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

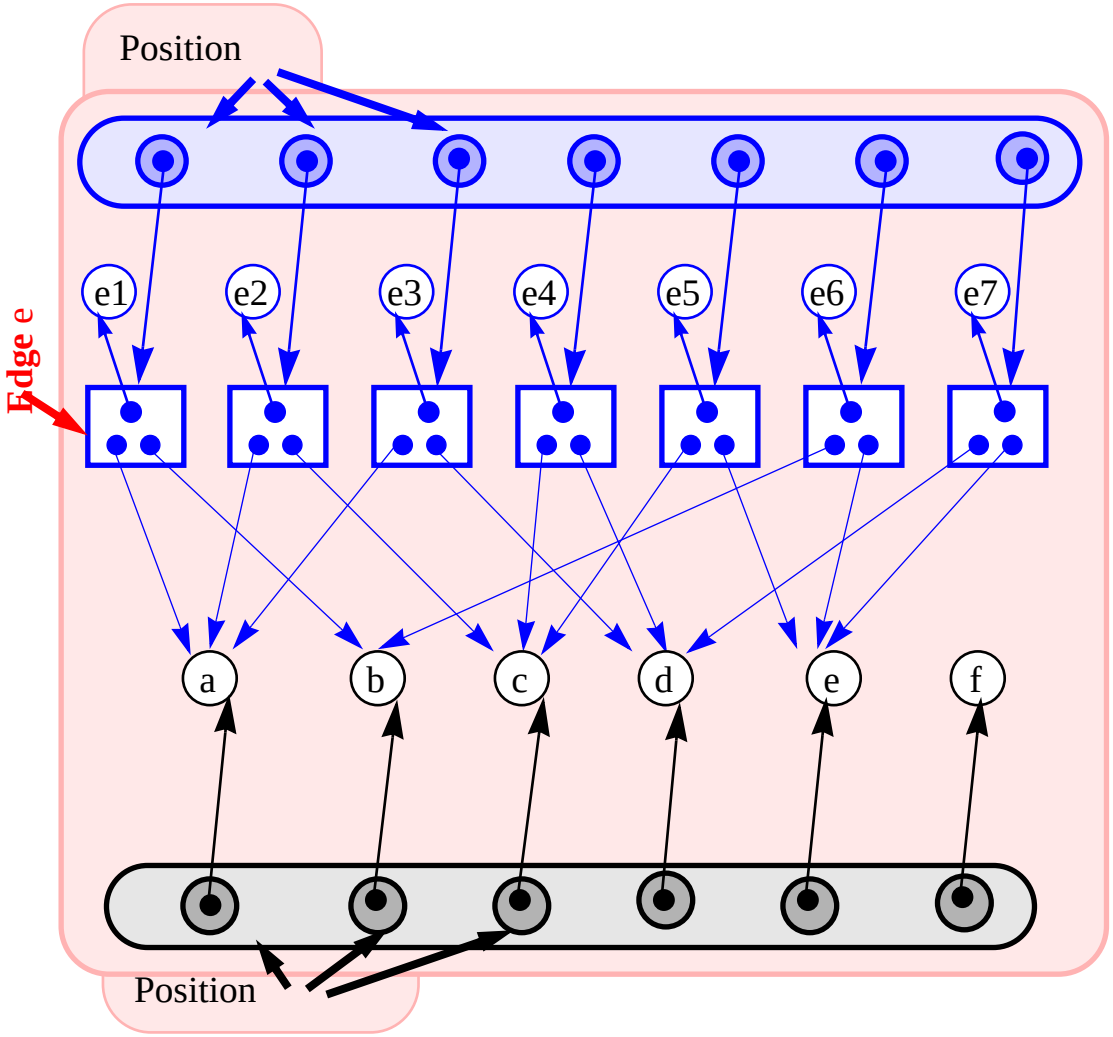
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

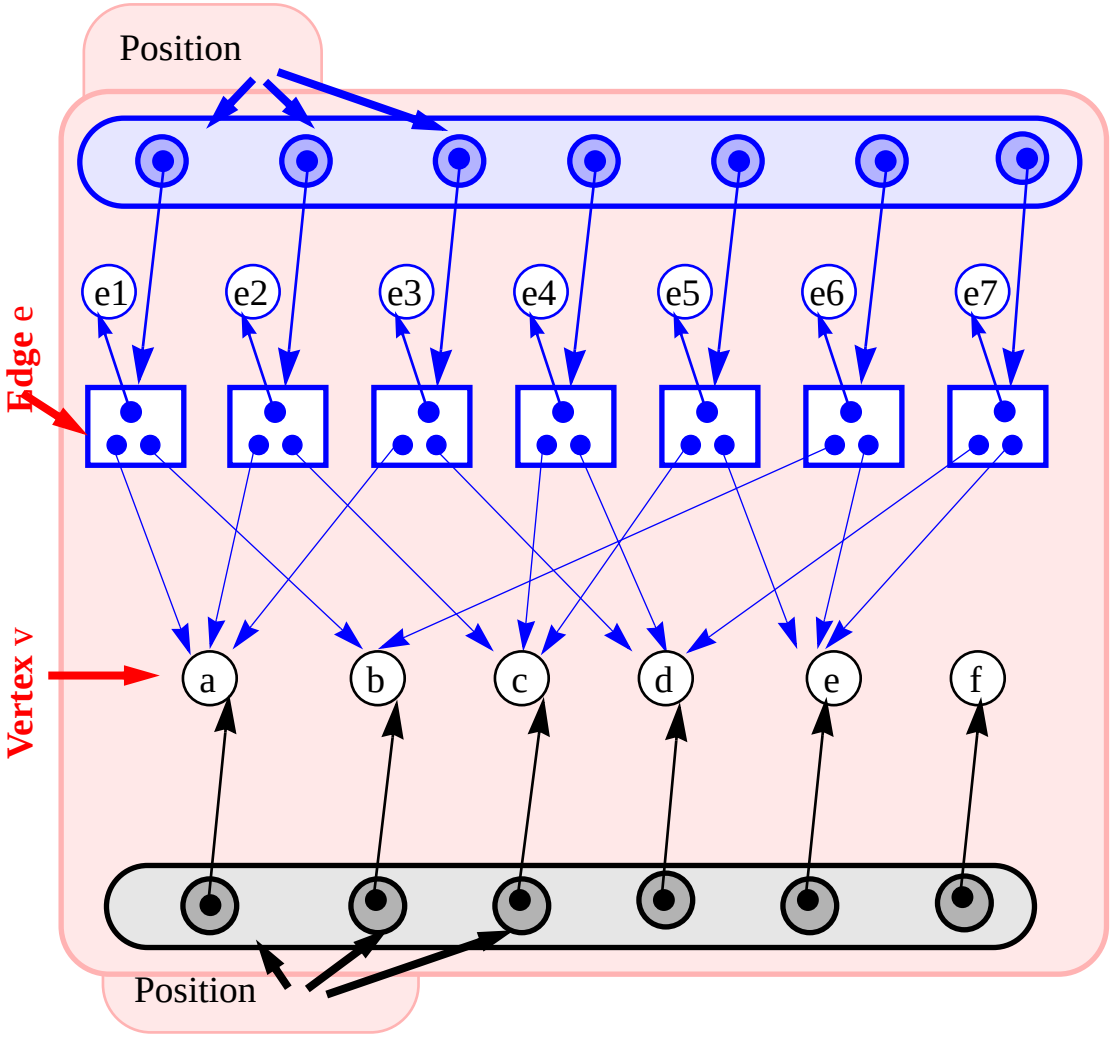
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

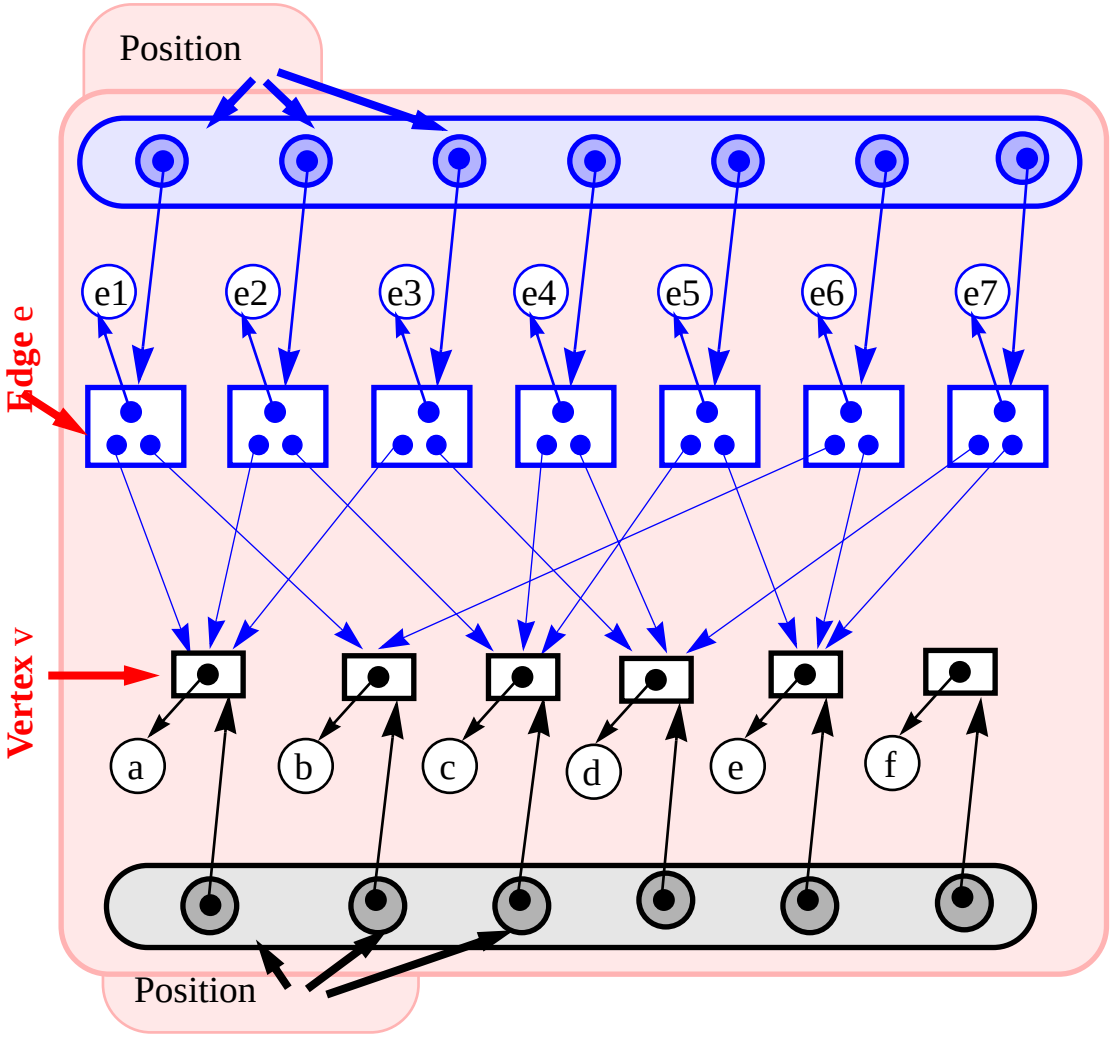
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

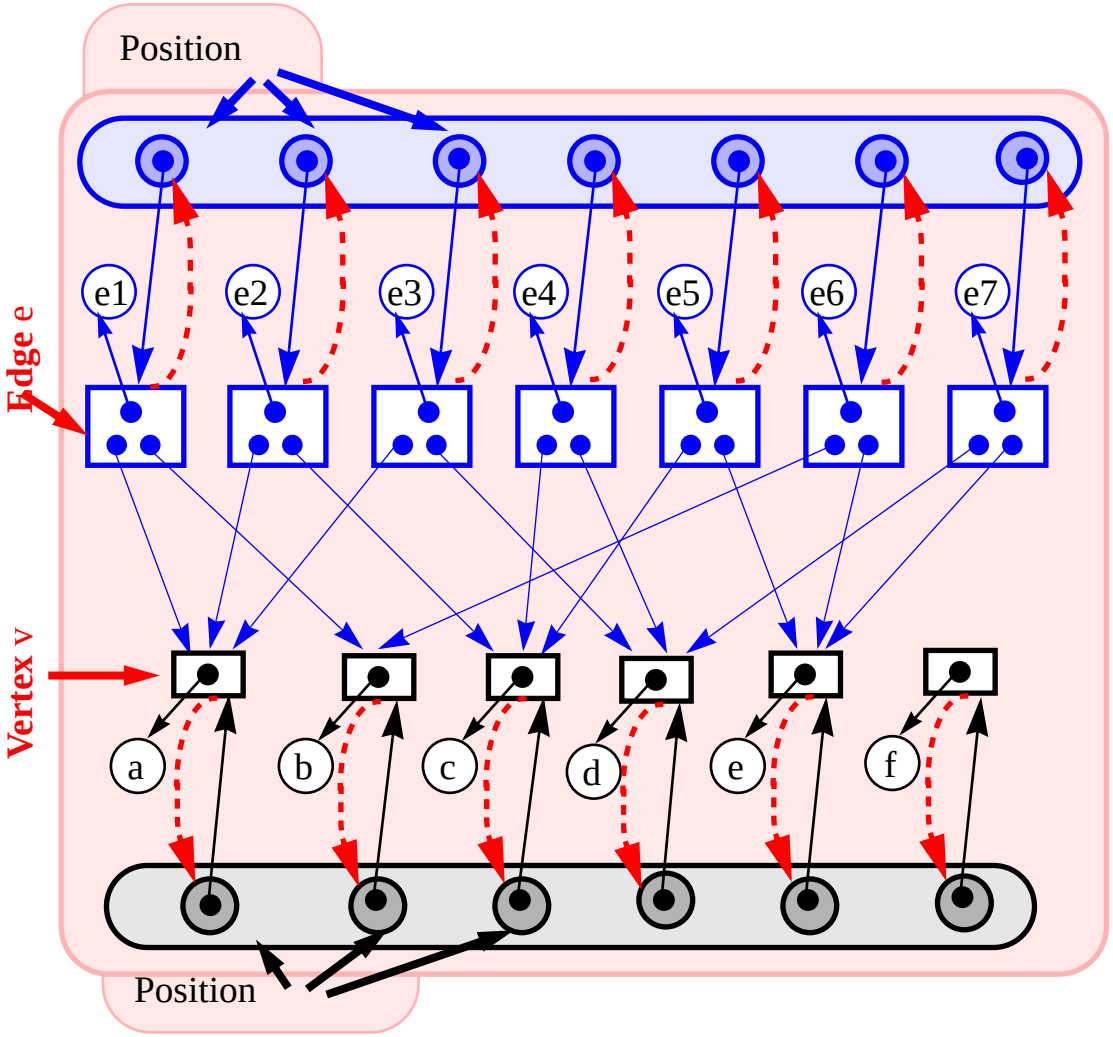
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

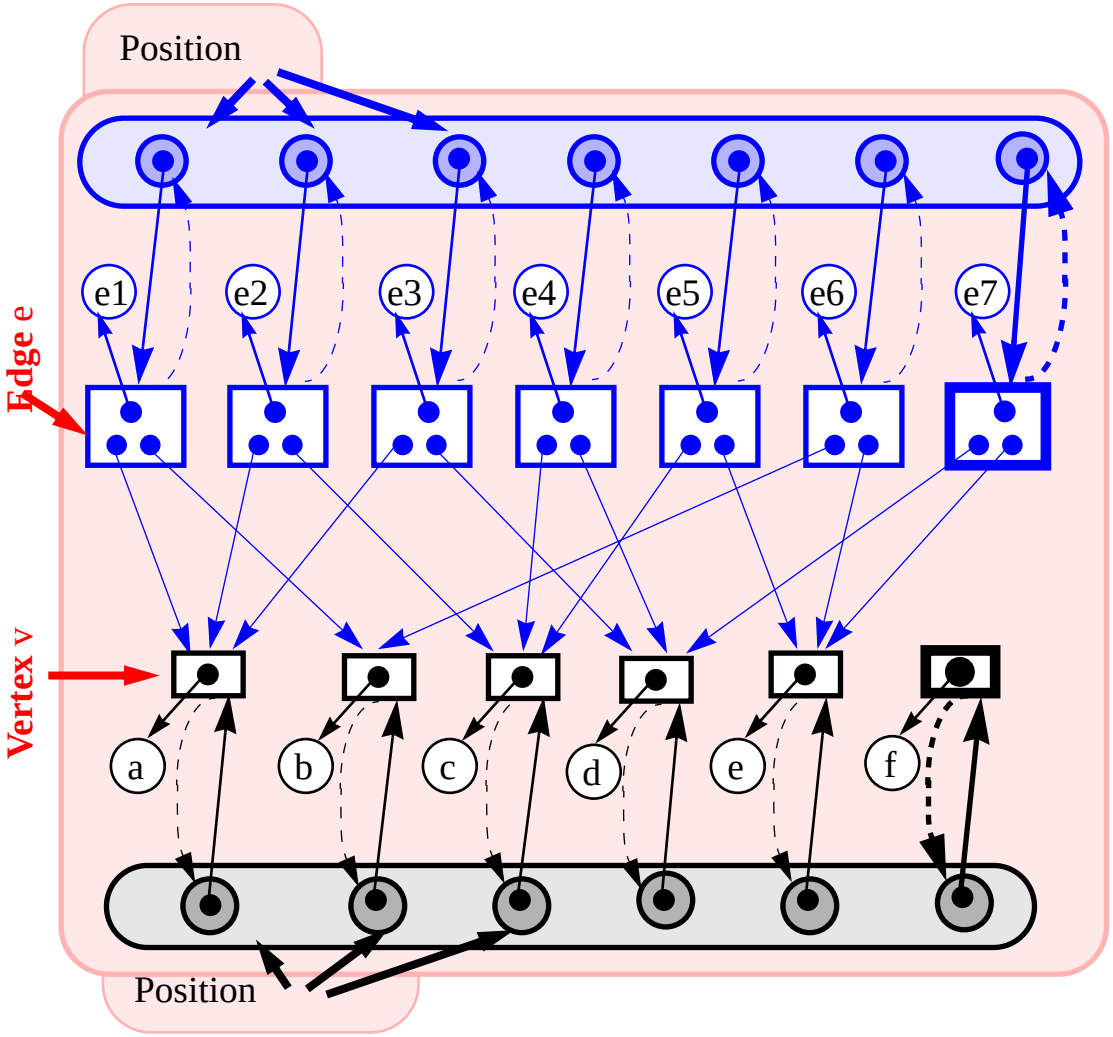
public ***GraphKL***(List **k**, **n**) { **kanter** = k; **noder** = n; }



public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

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



DataInvariant:

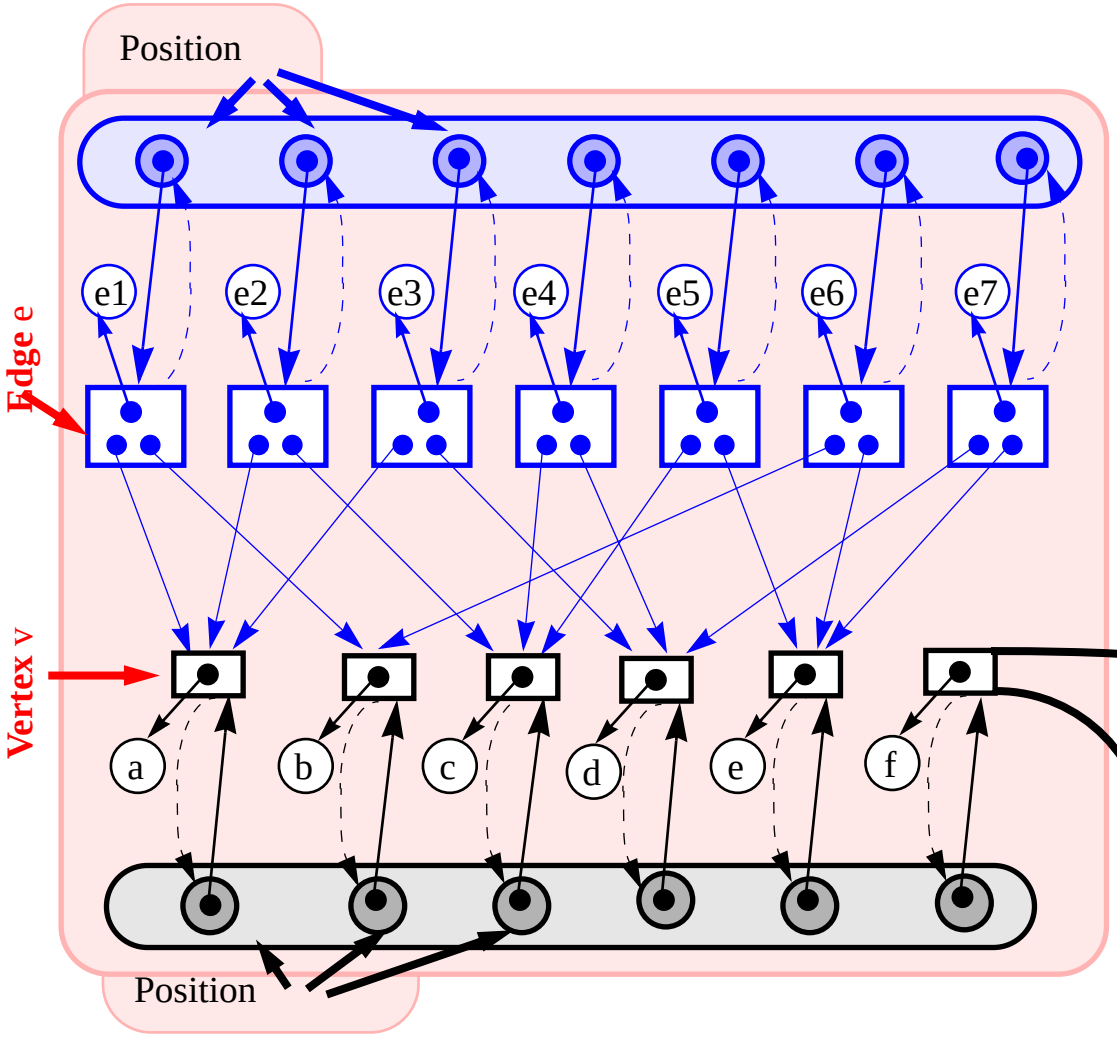
e . iKanter() . element() == e

n . iNoder() . element() == n

public class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

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



DataInvariant:

e . iKanter() . element() == e

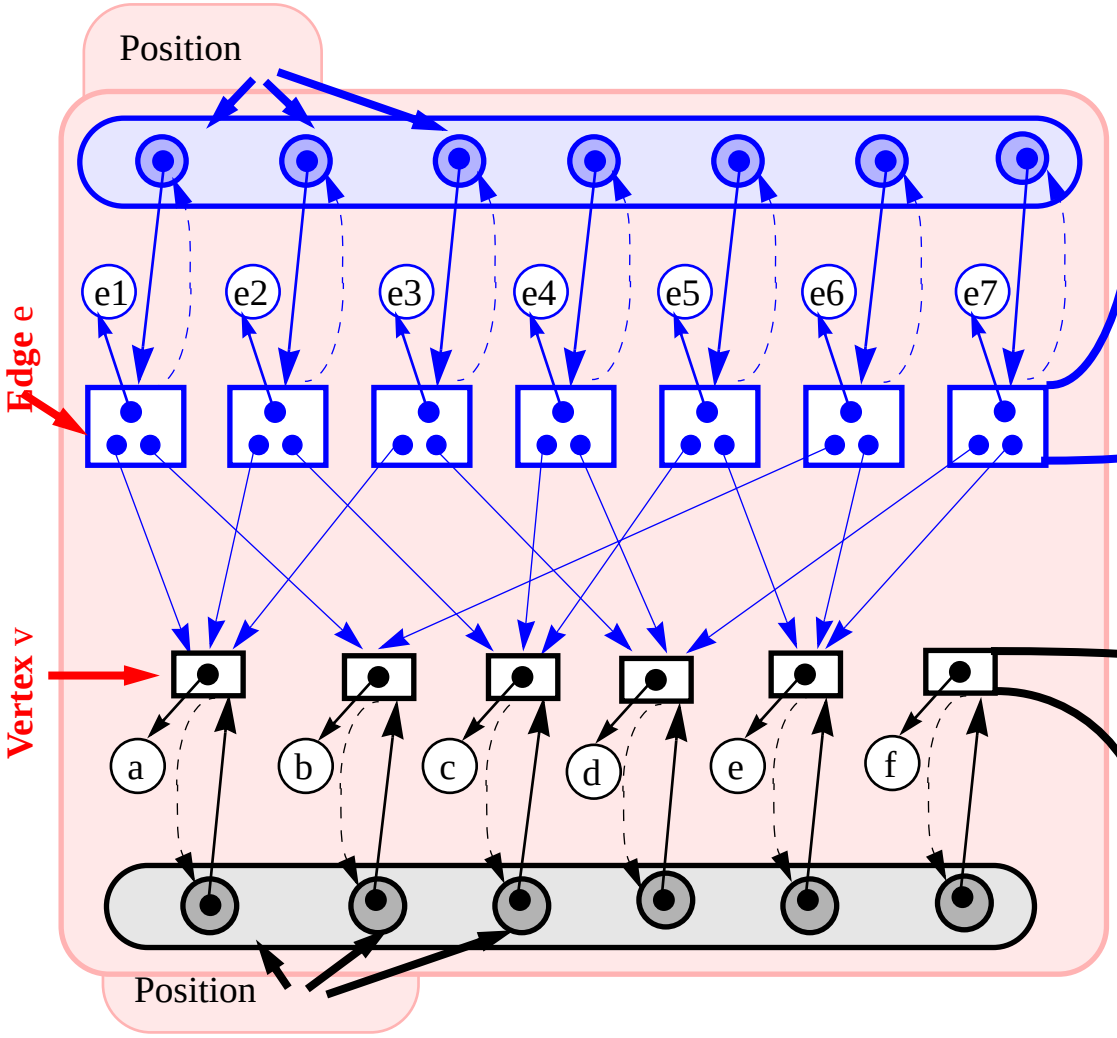
n . iNoder() . element() == n

```
class KLNode implements Vertex {
  protected Object elem; int deg=0;
  protected Position np;
  ----------------------------------
  public KLNode(Object o)  { elem=o;}
  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 class *GraphKL* implements *Graph* {

private List **kanter**, **noder**;

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




```
class KLEdge implements Edge {
  protected Object elem;
  protected Position kp;
  protected Vertex[] ee=new Vertex[2];
  public KLEdge (Vertex a,Vertex b,
                 Object o) {
    elem=o; ee[0]=a; ee[1]=b; }
  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:

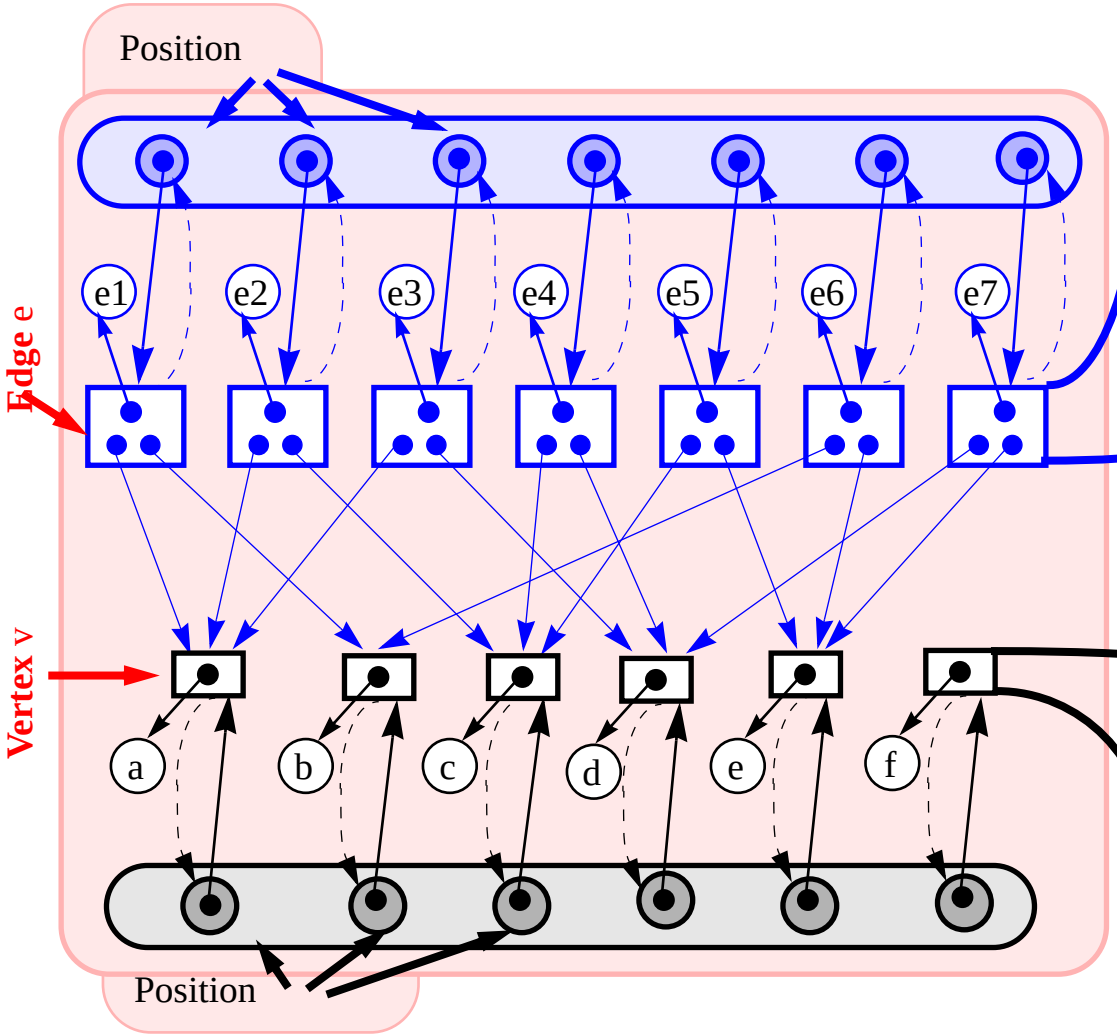
e . iKanter() . element() == e

n . iNoder() . element() == n

```
class KLNode implements Vertex {
  protected Object elem; int deg=0;
  protected Position np;
  public KLNode(Object o) { elem=o;}
  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 class *GraphKL* implements *Graph* {

```
private List kanter, noder;
public GraphKL(List k, n)   { kanter = k; noder = n; }
```




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

```
class KLEdge implements Edge {
  protected Object elem;
  protected Position kp;
  protected Vertex[] ee=new Vertex[2];
  public KLEdge (Vertex a,Vertex b,
                 Object o)  {
    elem=o; ee[0]=a; ee[1]=b; }
  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; }
  ... }
```

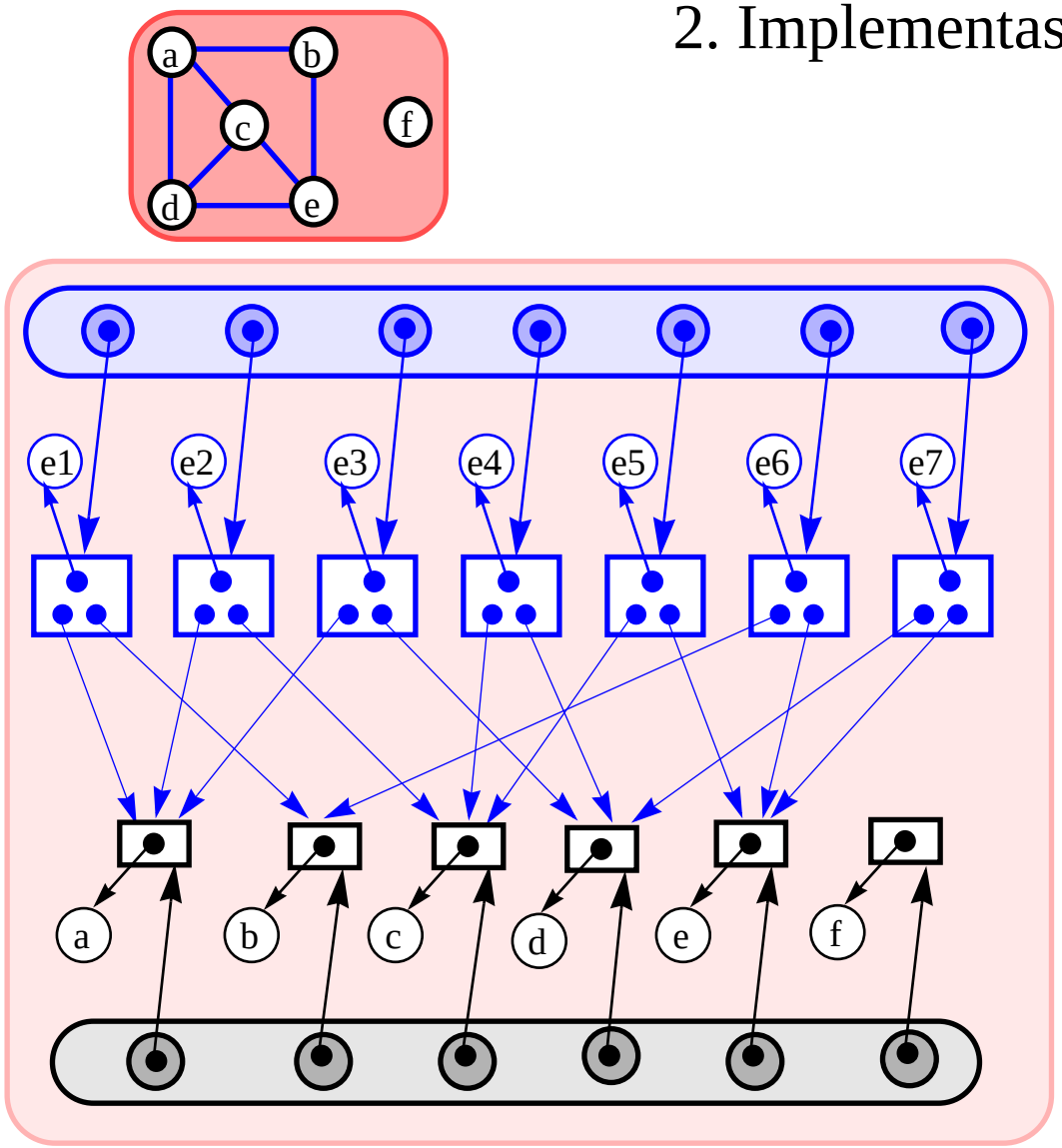
DataInvariant:

e . iKanter() . element() == e

n . iNoder() . element() == n

```
class KLNode implements Vertex {
  protected Object elem; int deg=0;
  protected Position np;
  public KLNode(Object o)  { elem=o;}
  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; }
... }
```

2. Implementasjon av Graph *med Kant-Liste*



endV(e), opposite(v,e),
degree(v) ..... $O(1)$

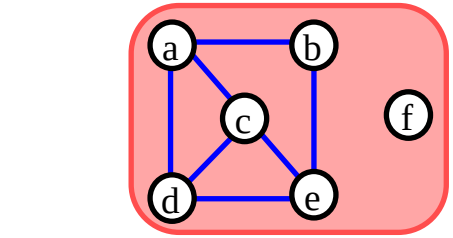
insertV(o), insertE(v,u,o),
removeE(e) ..... $O(1)$

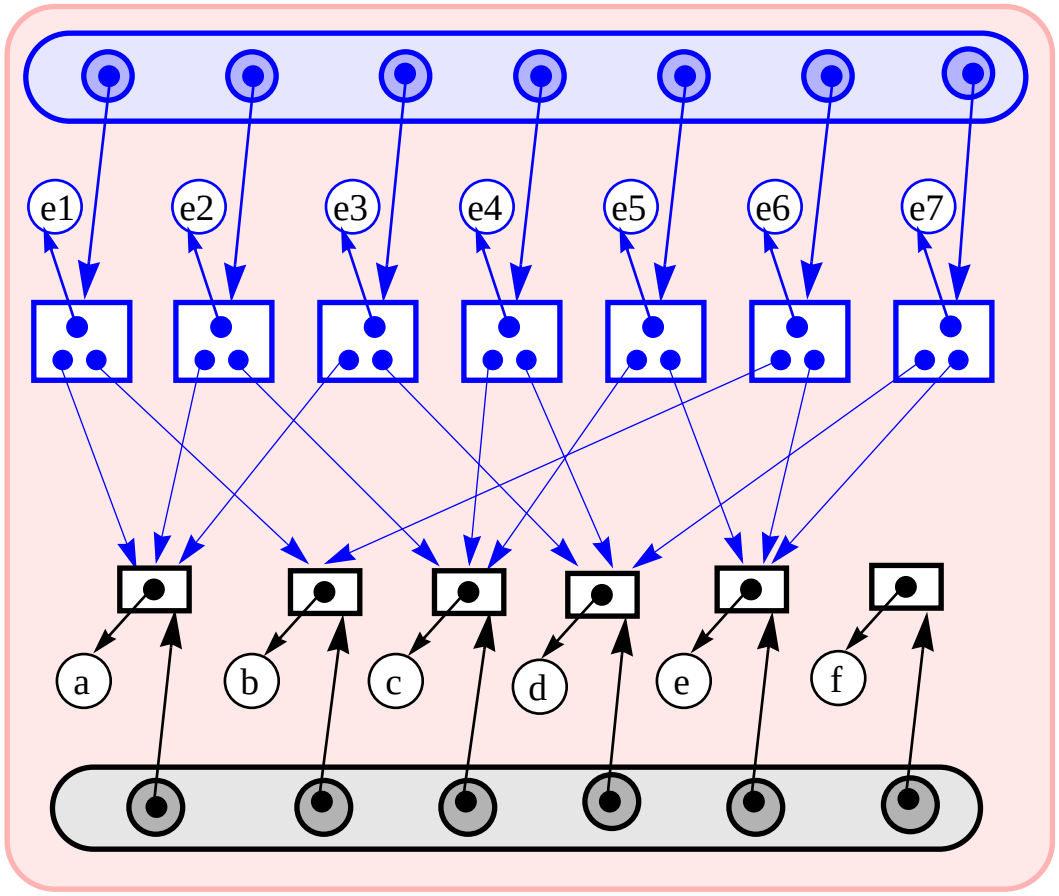
incidentE(v),
adjacentV(v) ..... $O(k)$

removeV(v) ..... $O(k)$

areAdjacent(v,u) ..... $O(k)$

2. Implementasjon av Graph *med Nabo-Liste*



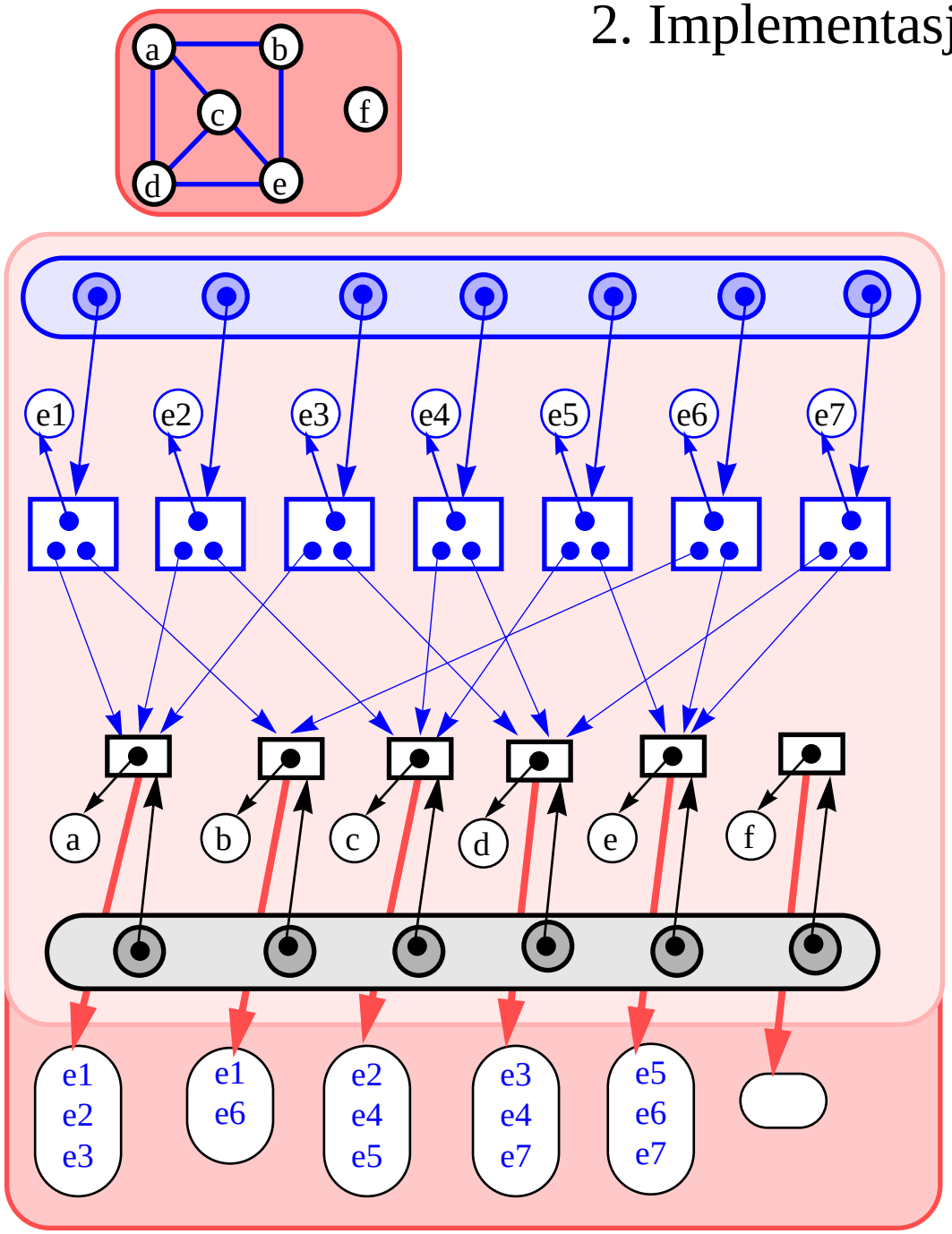


er som Kant-Liste men i tillegg

- hver node v har en samling Inc(v) med sine (incident) kanter

endV(e), opposite(v,e),
degree(v) ..... *O*(**1**)
insertV(o), insertE(v,u,o),
removeE(e) ..... *O*(**1**)
incidentE(v),
adjacentV(v) ..... *O*(**k**)
removeV(v) ..... *O*(**k**)
areAdjacent(v,u) ..... *O*(**k**)

2. Implementasjon av Graph *med Nabo-Liste*

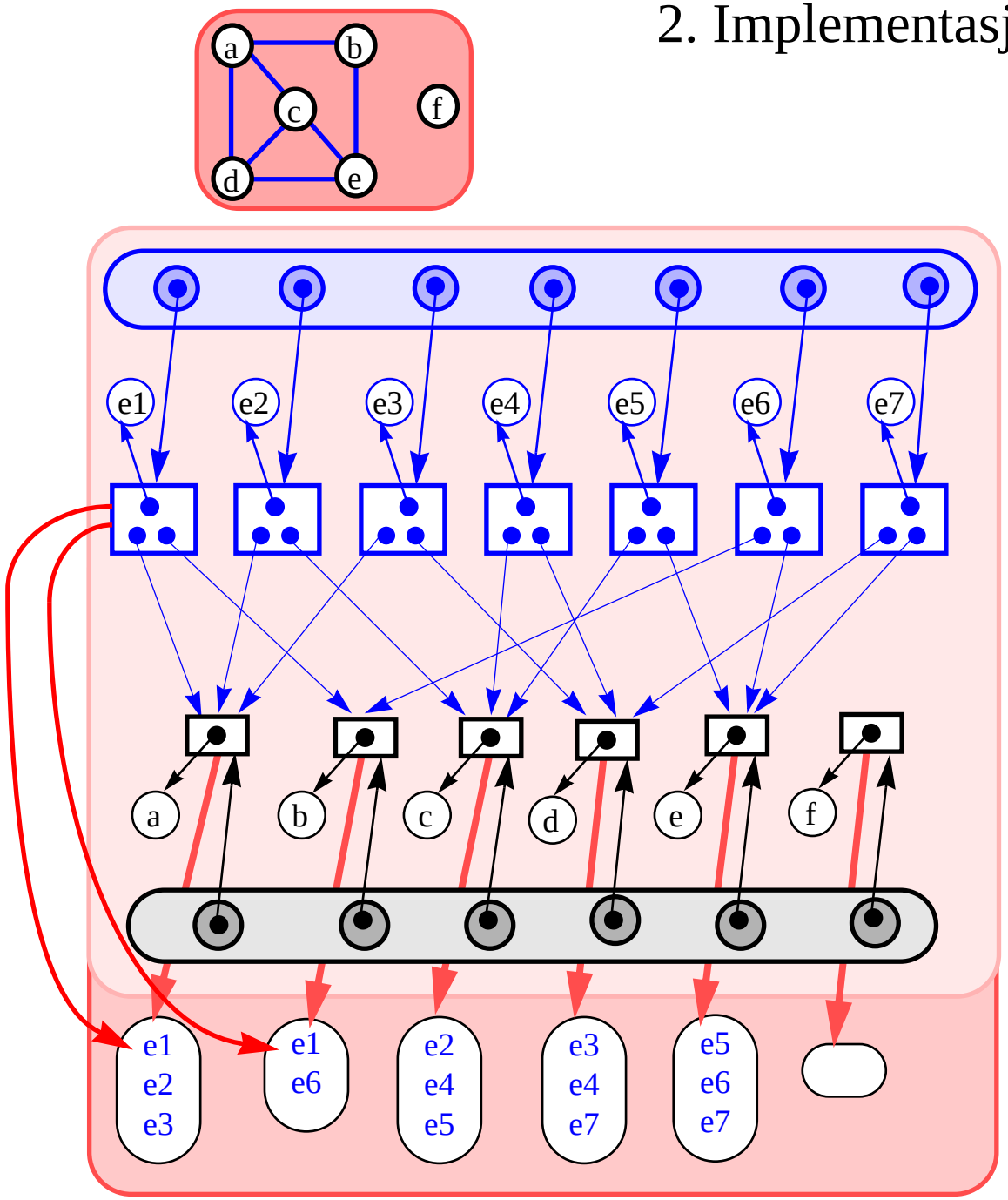


er som Kant-Liste men i tillegg

- hver node v har en samling Inc(v) med sine (incident) kanter

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

2. Implementasjon av Graph *med Nabo-Liste*

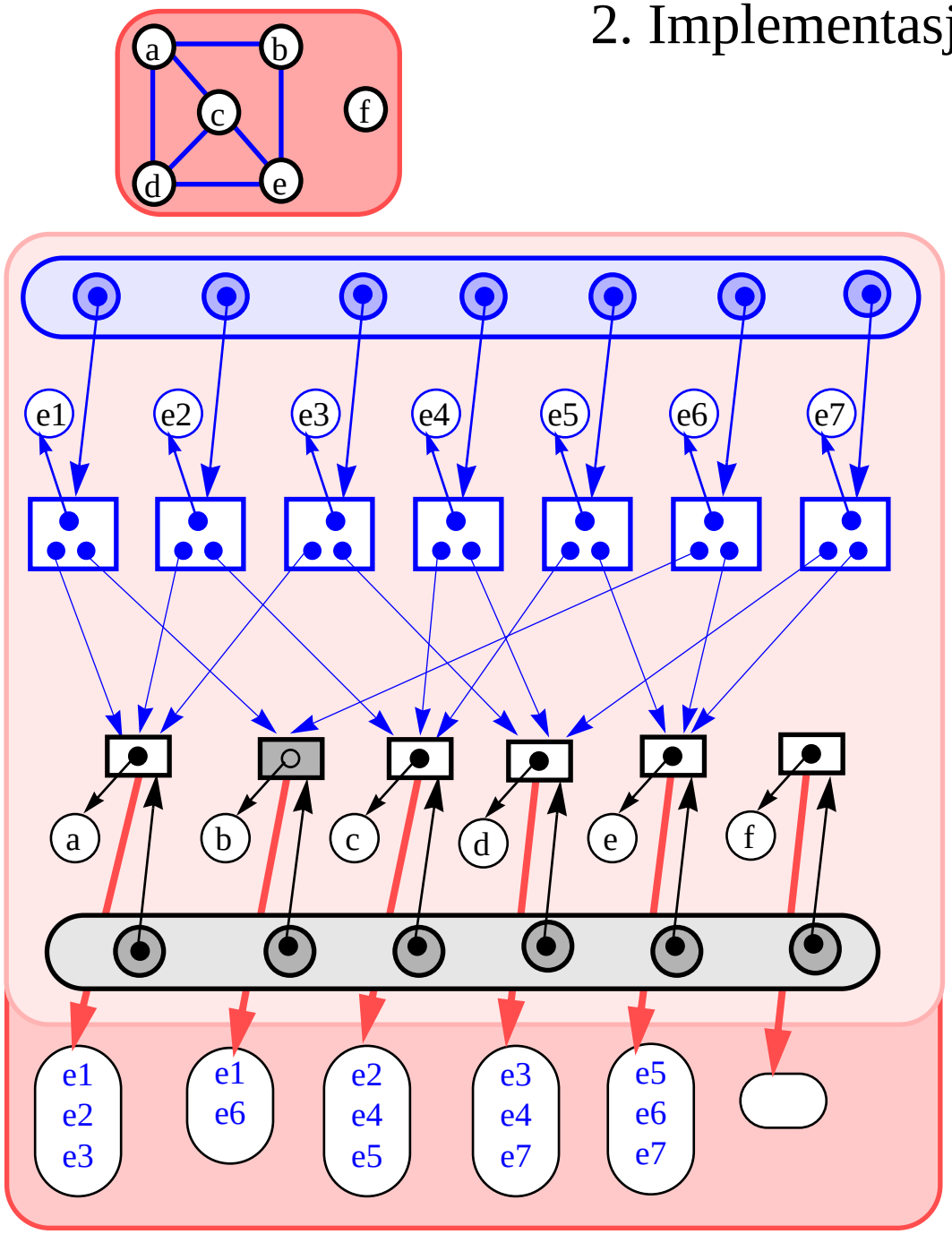


er som Kant-Liste men i tillegg

- 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)	$O(\mathbf{1})$
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)	$O(\mathbf{k})$

2. Implementasjon av Graph *med Nabo-Liste*



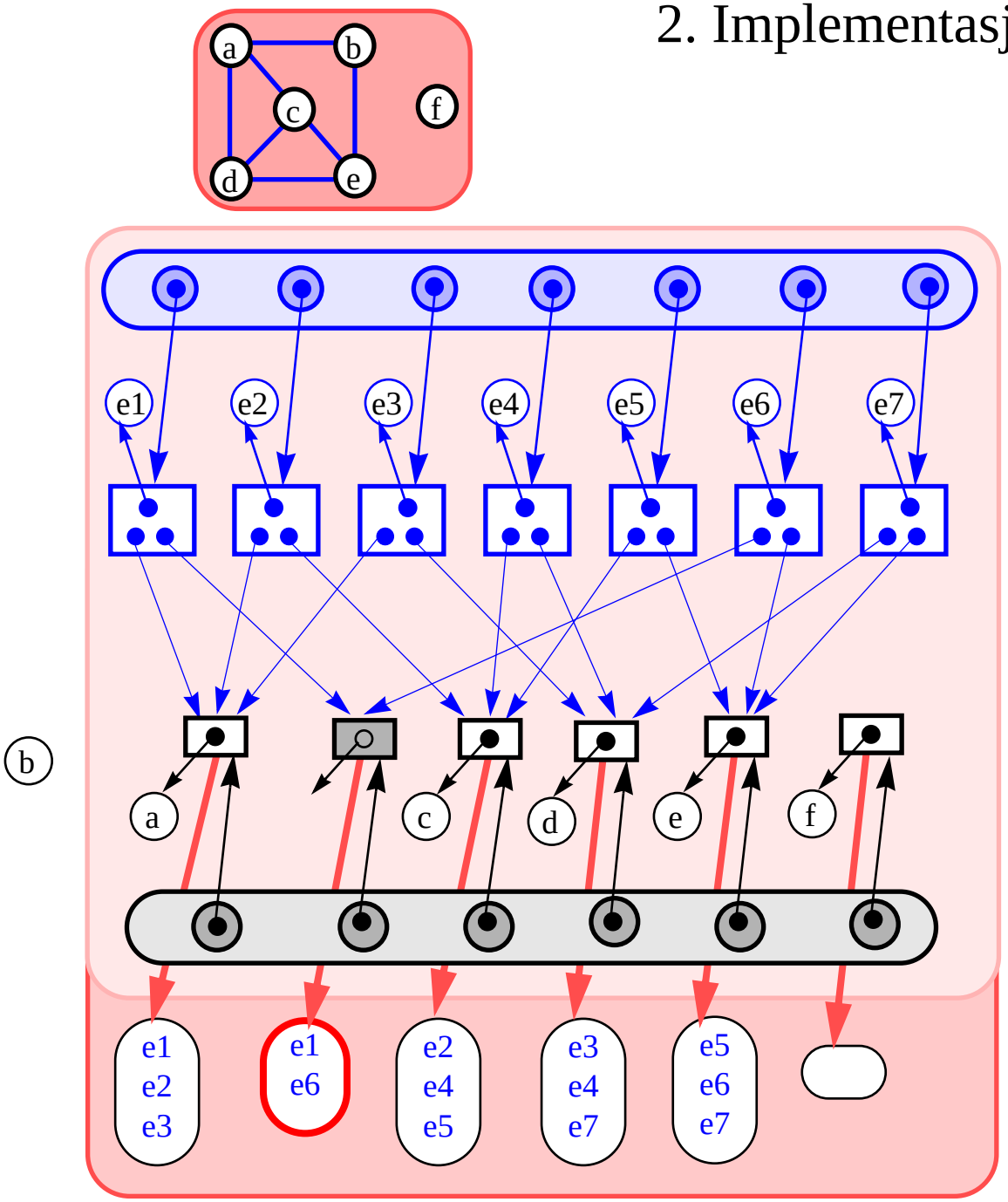
er som Kant-Liste men i tillegg

- 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**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)
removeV(v)	*O*(**k**)
areAdjacent(v,u)	*O*(**k**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret = v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



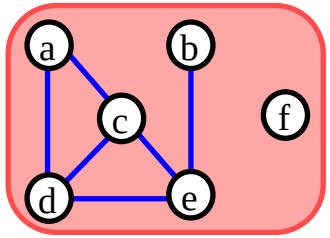
er som Kant-Liste men i tillegg

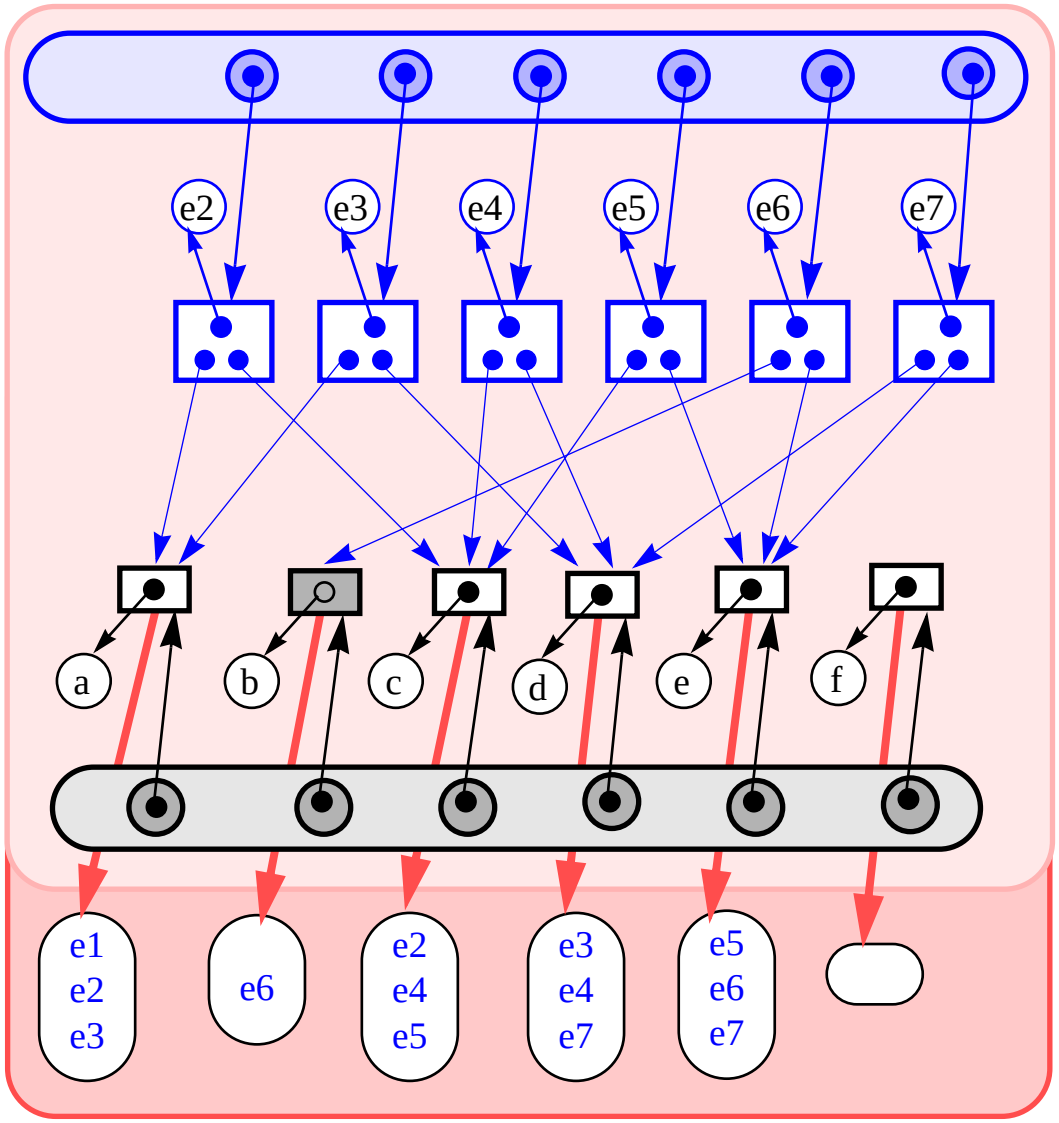
- hver node v har en samling Inc(v) med sine (incident) kanter
- (hver kant (u,v) holder en referanse til sin posisjon i Inc(v) og Inc(u))

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

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*





b

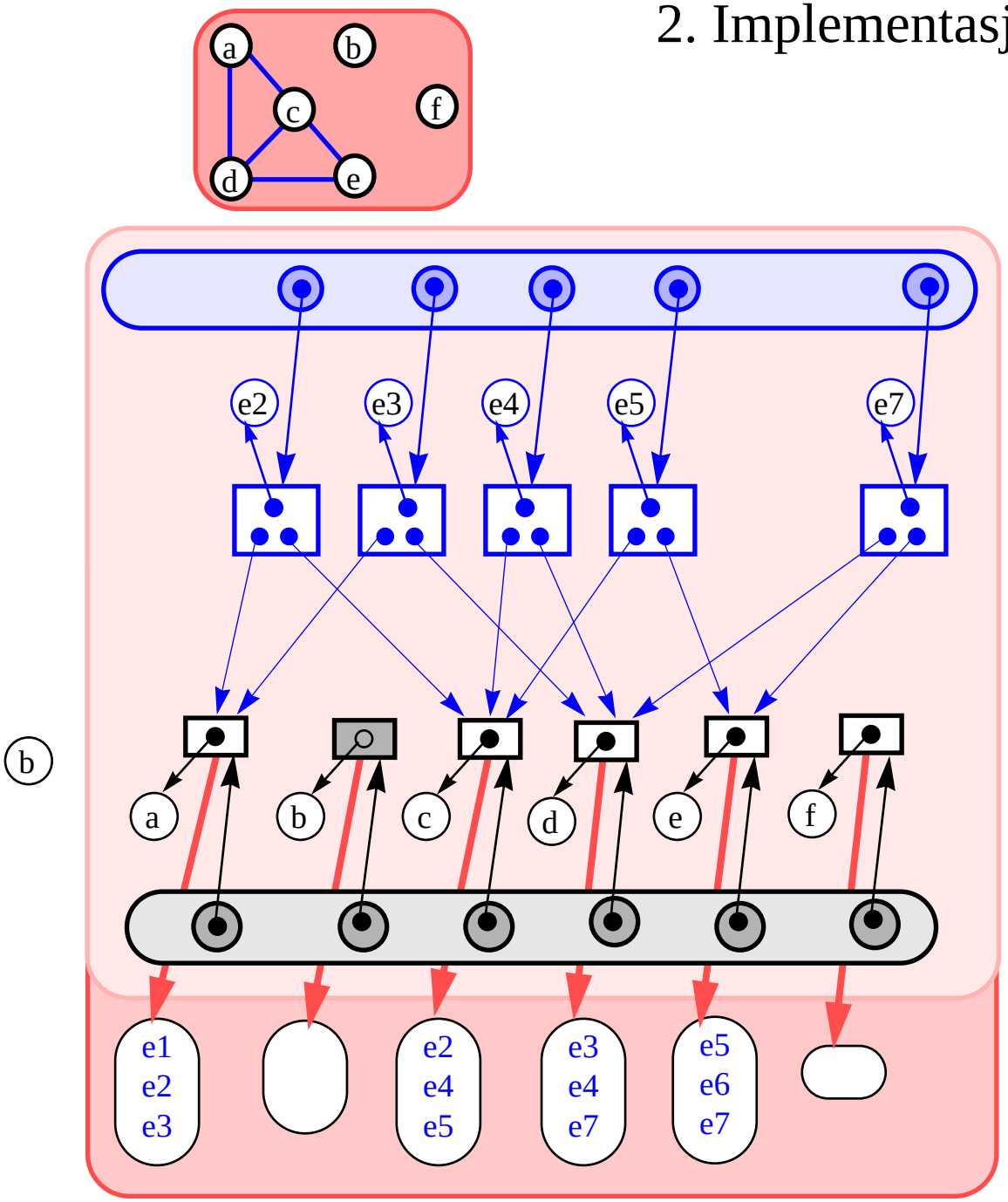
er som Kant-Liste men i tillegg

- 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**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)
removeV(v)	*O*(**k**)
areAdjacent(v,u)	*O*(**k**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



b

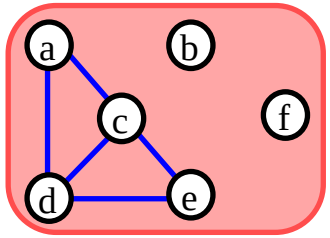
er som Kant-Liste men i tillegg

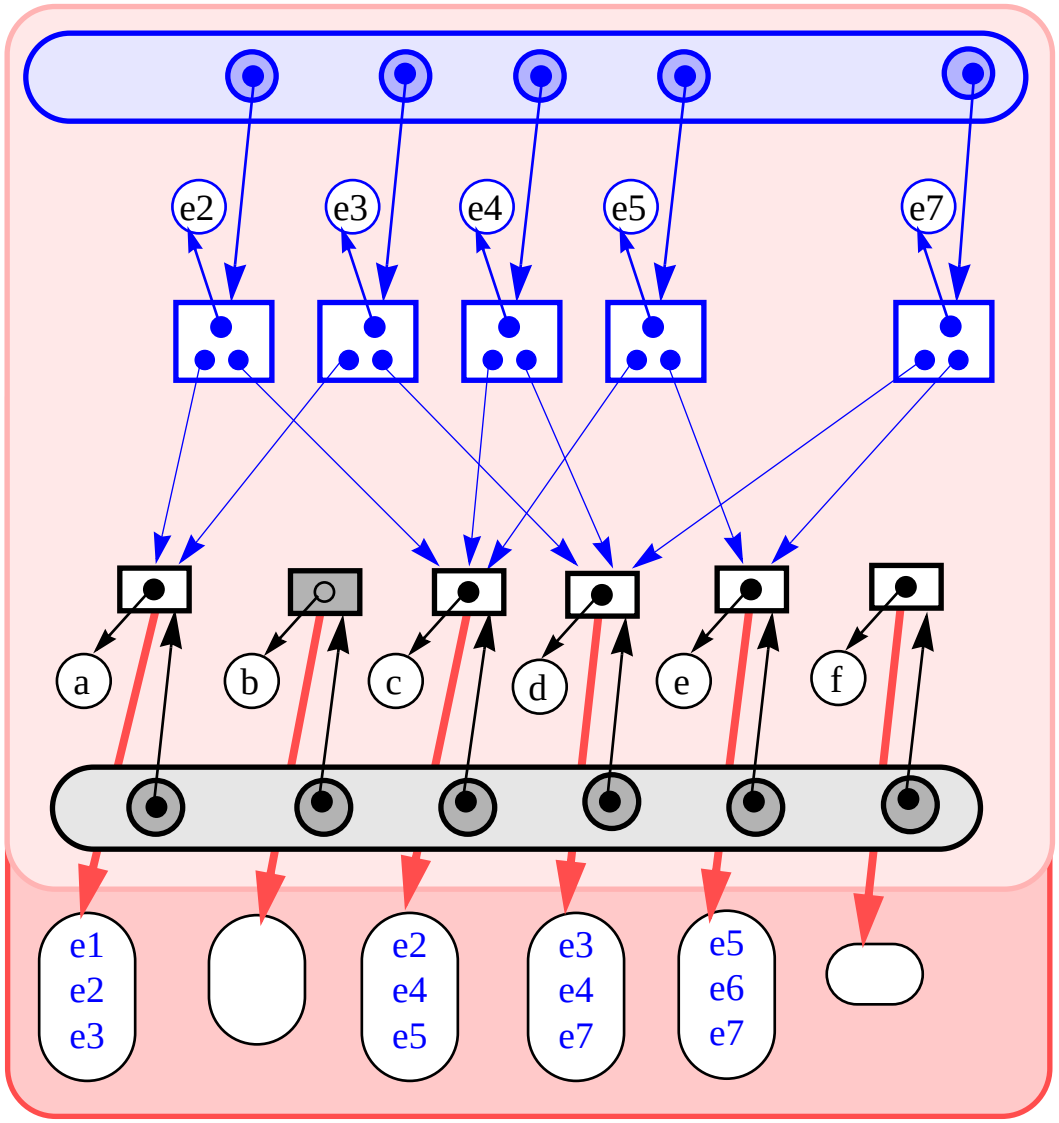
- hver node v har en samling Inc(v) med sine (incident) kanter
- (hver kant (u,v) holder en referanse til sin posisjon i Inc(v) og Inc(u))

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

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*





b

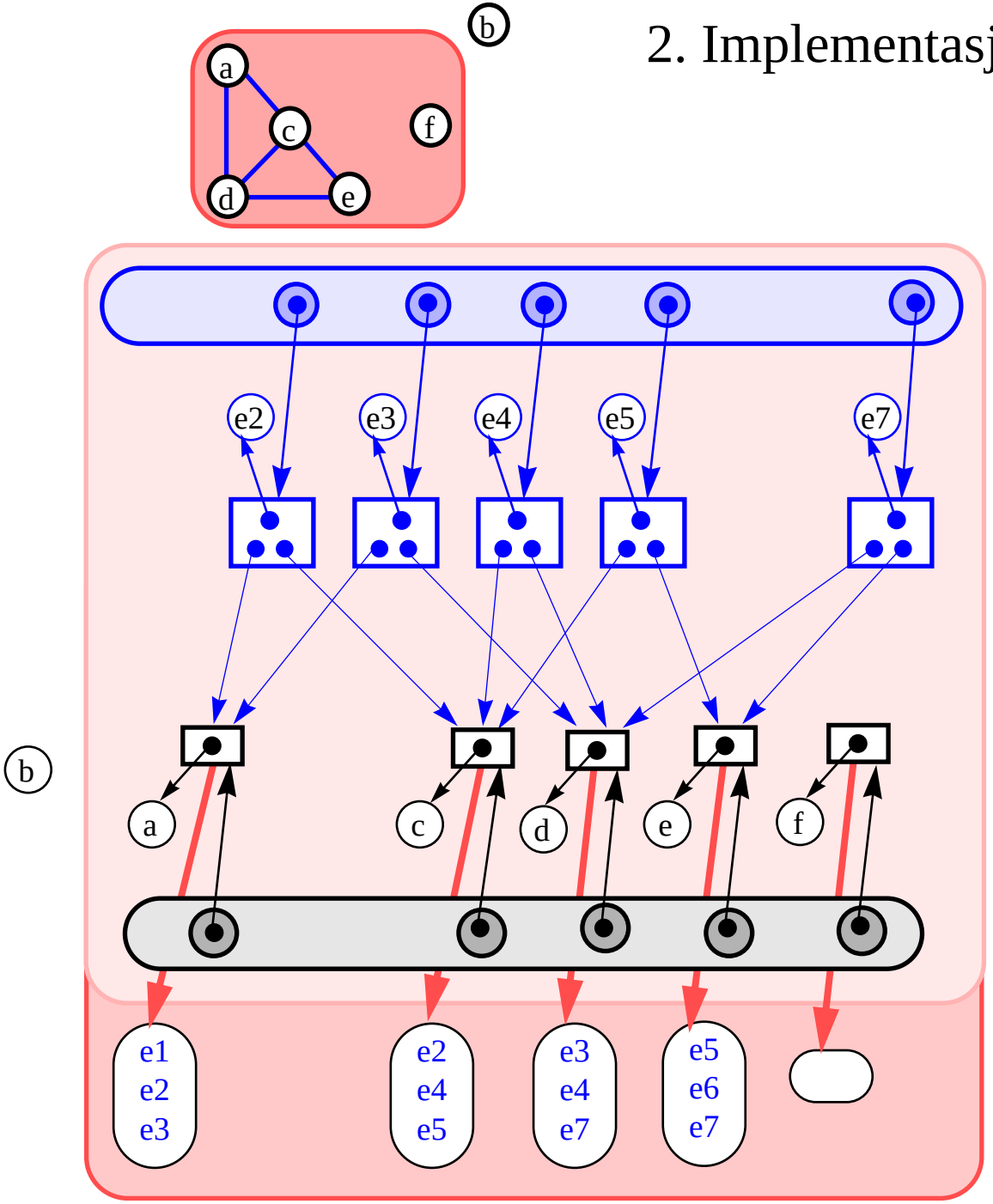
er som Kant-Liste men i tillegg

- 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**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)
removeV(v)	*O*(**k**)
areAdjacent(v,u)	*O*(**k**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



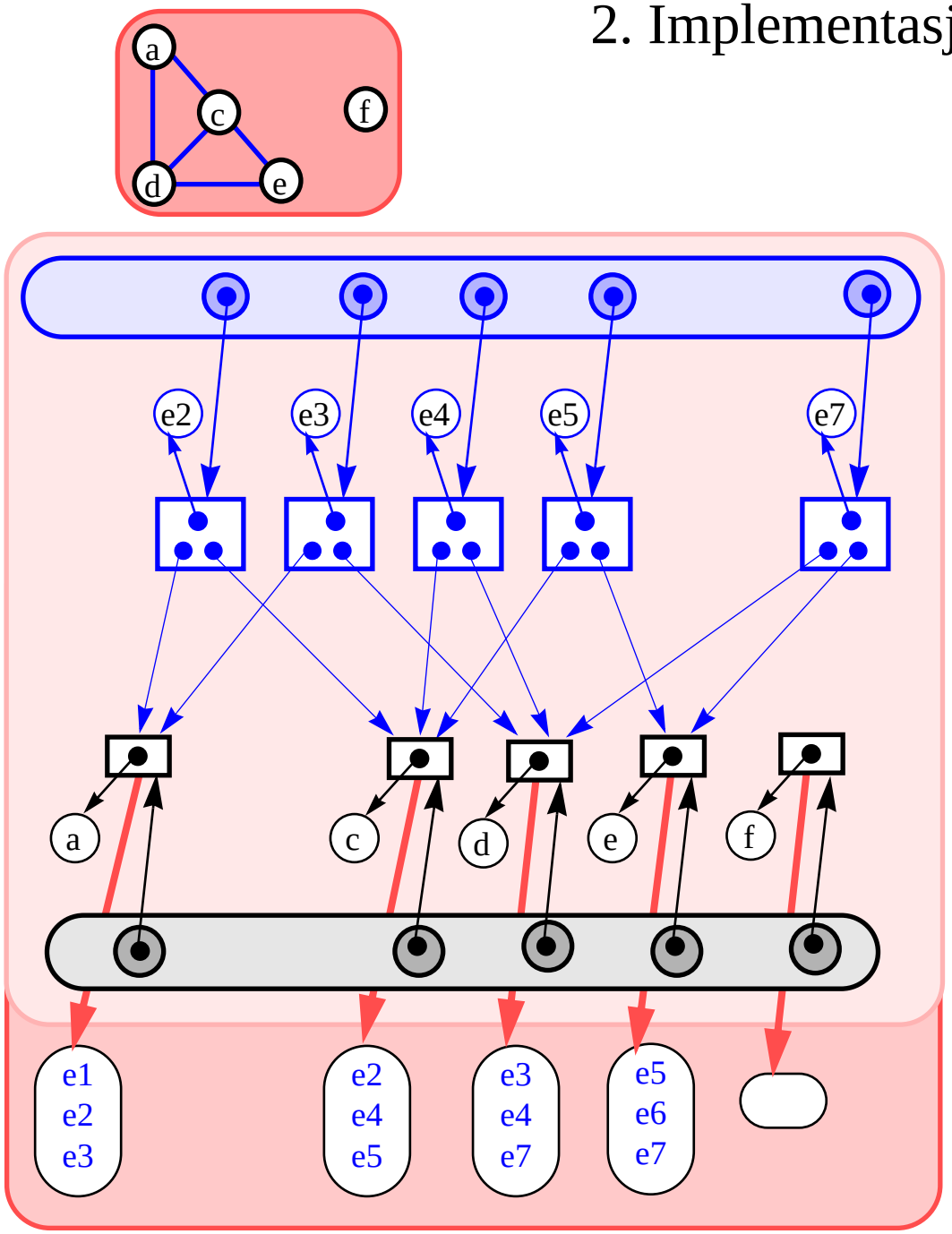
er som Kant-Liste men i tillegg

- 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**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)
removeV(v)	*O*(**k**)
areAdjacent(v,u)	*O*(**k**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    while (k.hasMoreElements() ) {
      NLNode e= (NLNode)k.nextElemet();
      if (e.has(v)) removeEdge(e);  }
    noder.remove( ((NLNode)v).iNoder() );
    return ret;
  } else throw new InvalidPosExc(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



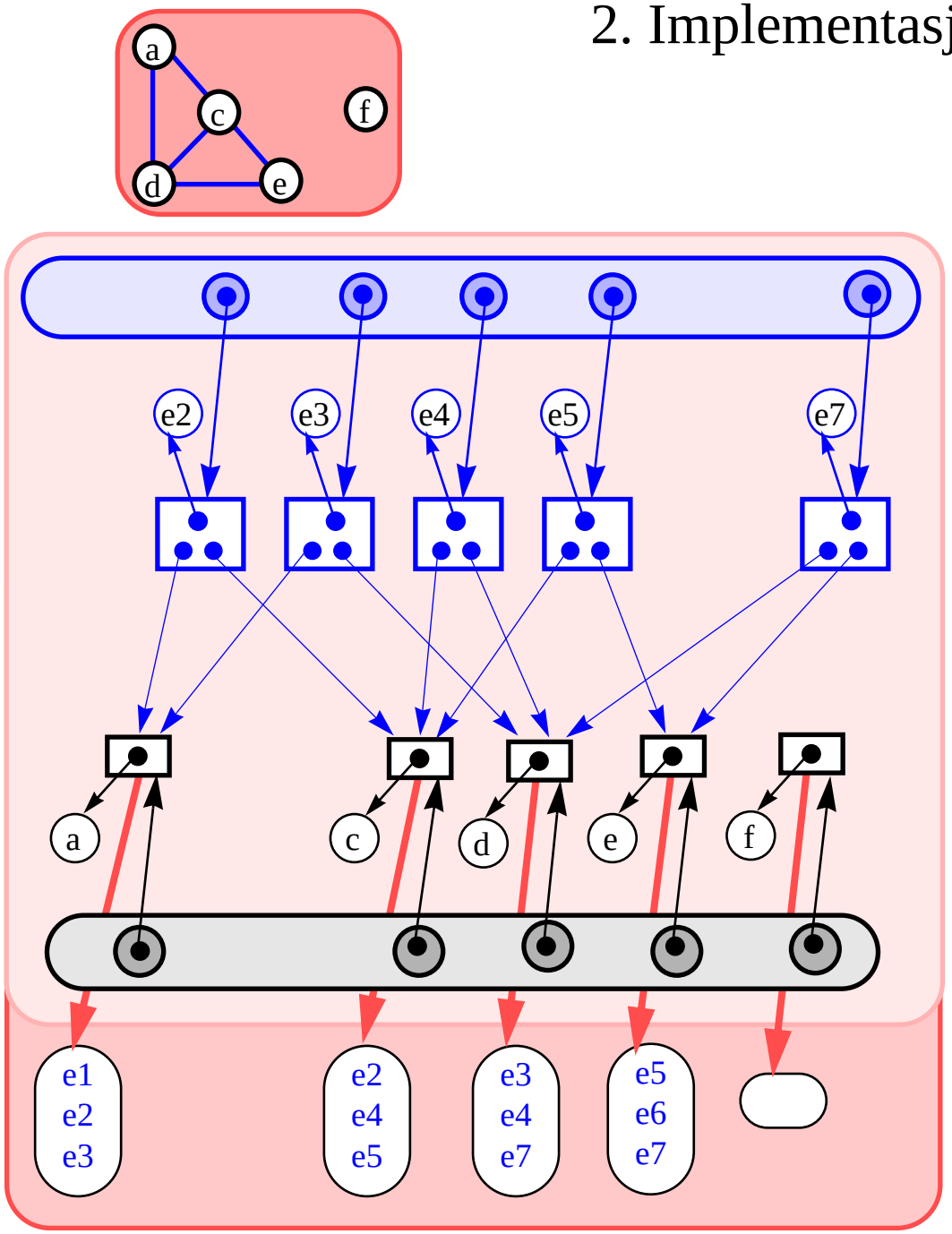
er som Kant-Liste men i tillegg

- 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**)
insertV(o), insertE(v,u,o), removeE(e)	*O*(**1**)
incidentE(v), adjacentV(v)	*O*(**k**)
removeV(v)	*O*(**k**)
areAdjacent(v,u)	*O*(**k**)

```
Object removeVertex(Vertex v) {
  if (ok(v)) {  Object ret= v.element();
    Iterator k= Inc( (NLNode) v ) ;
    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(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



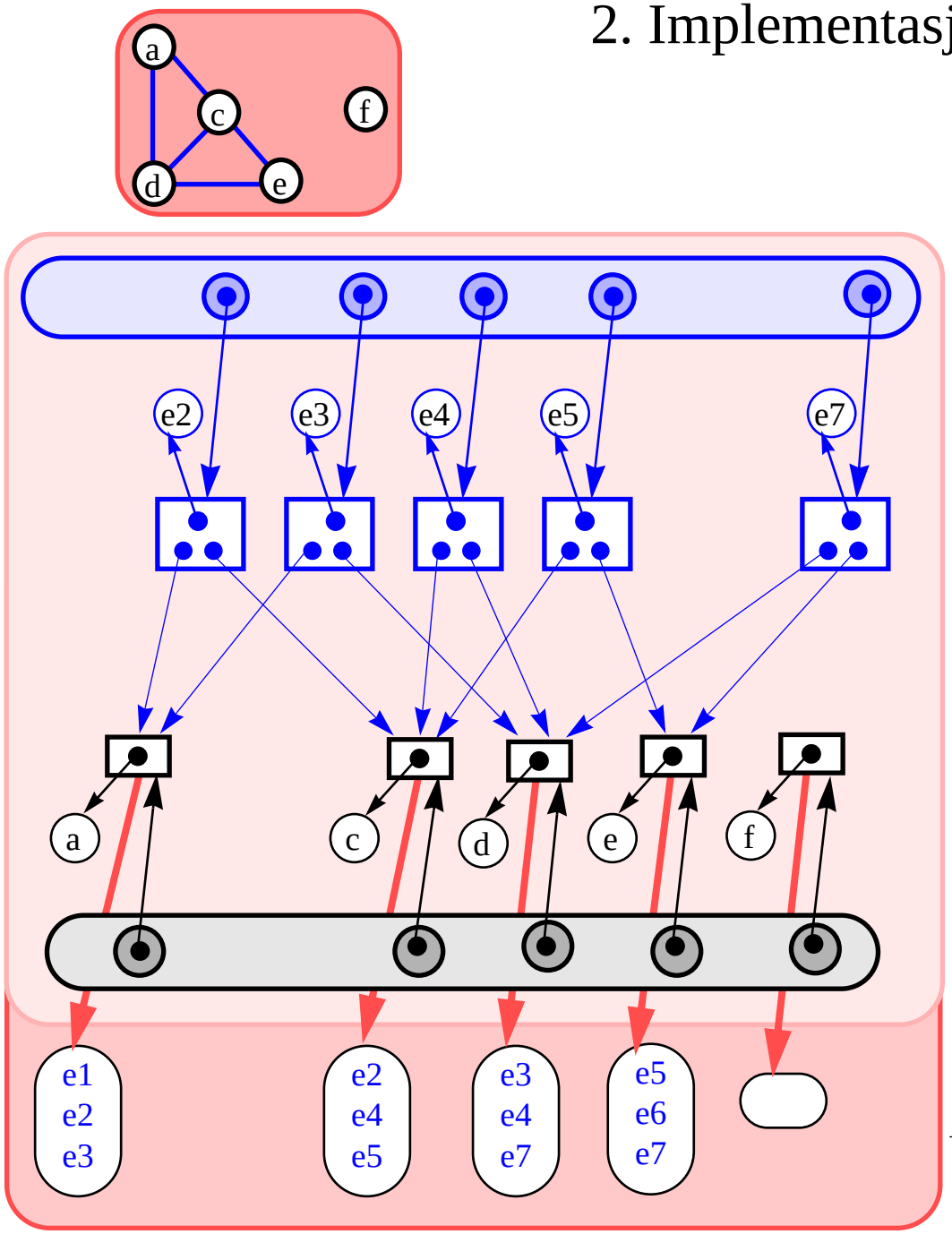
er som Kant-Liste men i tillegg

- 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();
    Iterator k= Inc( (NLNode) v ) ;
    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(“”);  }
```

2. Implementasjon av Graph *med Nabo-Liste*



er som Kant-Liste men i tillegg

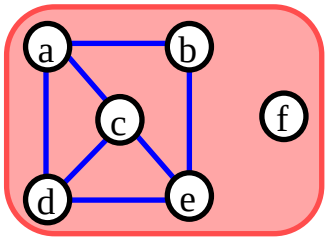
- hver node v har en samling Inc(v) med sine (incident) kanter
- (hver kant (u,v) holder en referanse til sin posisjon i Inc(v) og Inc(u))

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();
    Iterator k= Inc( (NLNode) v ) ;
    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

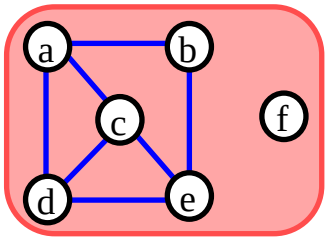
3. Implementasjon av Graph *med Nabo-Matrise*



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

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

3. Implementasjon av Graph *med Nabo-Matrise*



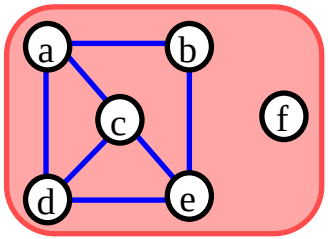
Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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

3. Implementasjon av Graph *med Nabo-Matrise*

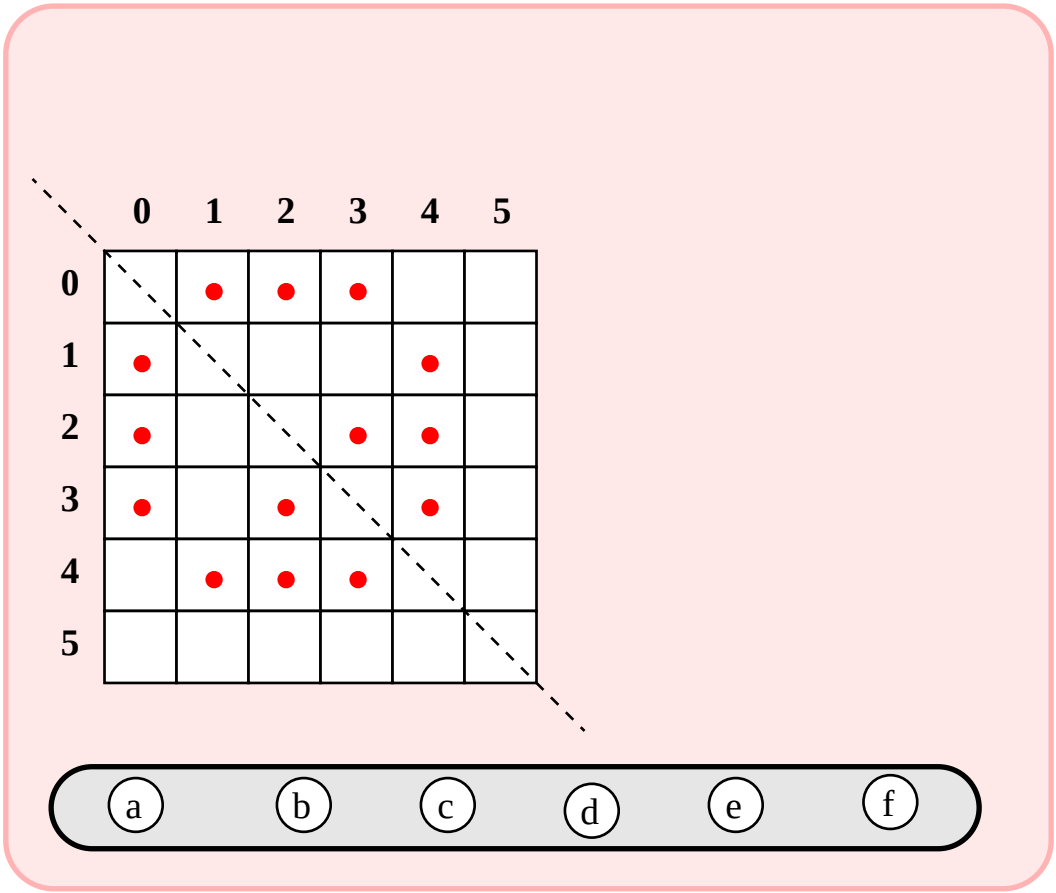


Enumerer alle noder fra 0...n-1:

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

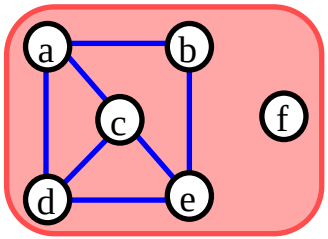
i en **n** x**n** matrise **boolean A[n][n]**

- **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



plass forbruk $O(n^2)$

3. Implementasjon av Graph *med Nabo-Matrise*

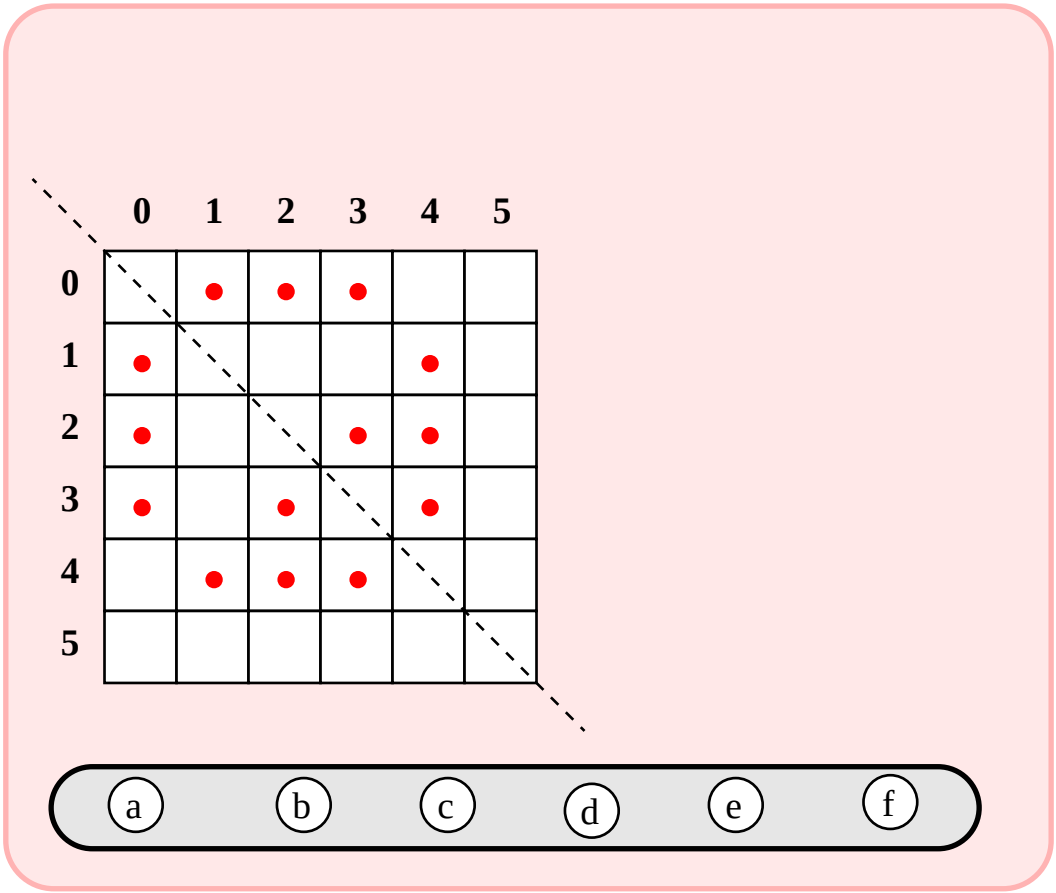


Enumerer alle noder fra 0...n-1:

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

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

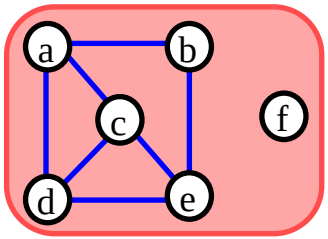
- **A[i][j]** == **true** hviss G har en kant e=(i,j)
- **A[i][j]** == **false** hviss (i,j) ikke er en kant i G



plass forbruk $O(n^2)$

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

3. Implementasjon av Graph *med Nabo-Matrise*

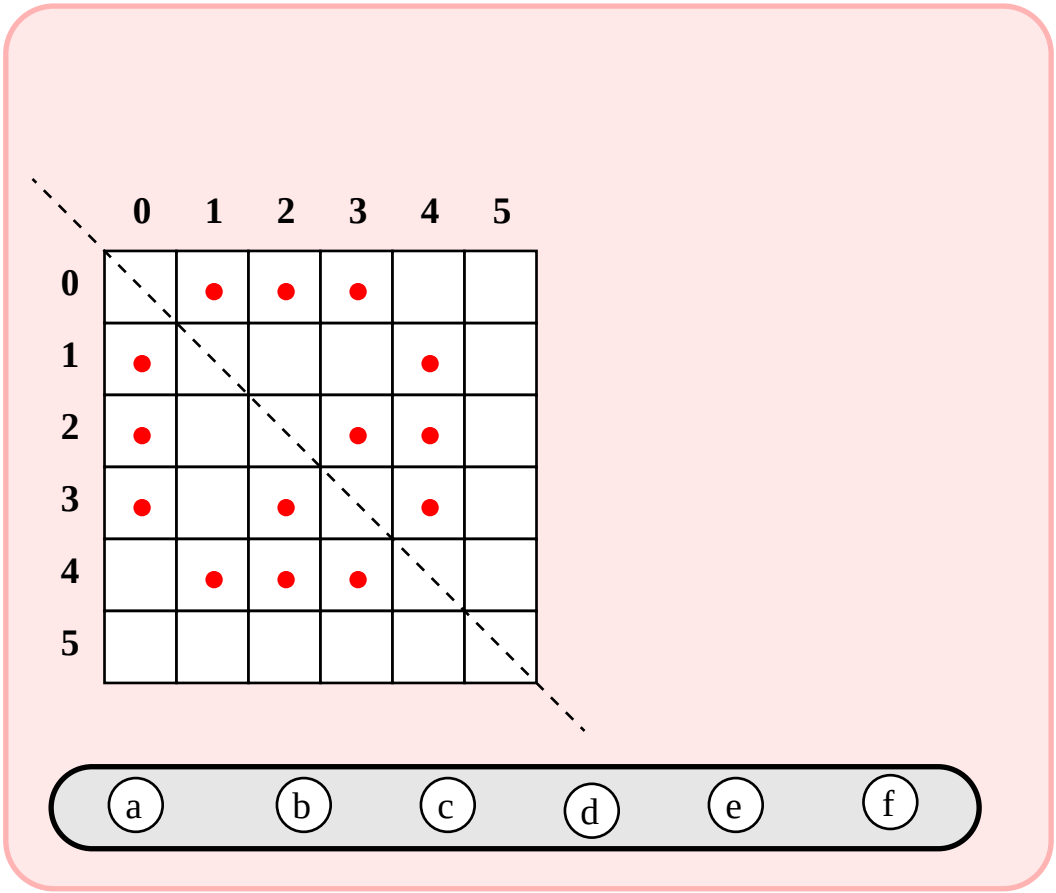


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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



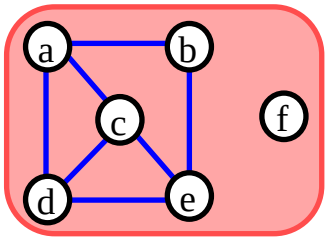
plass forbruk $O(n^2)$

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

DataInvariant for ikke-rettet graf:

A [**i**] [**k**] == **A** [**k**] [**i**]

3. Implementasjon av Graph *med Nabo-Matrise*

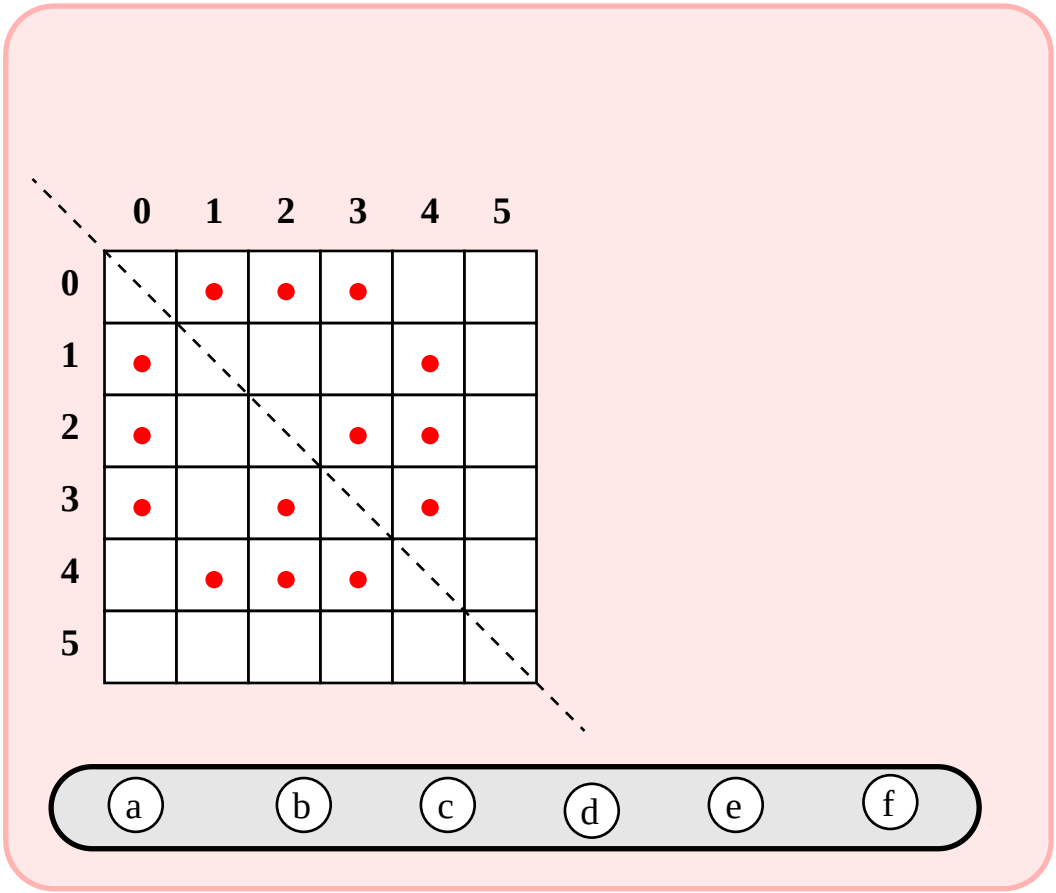


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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

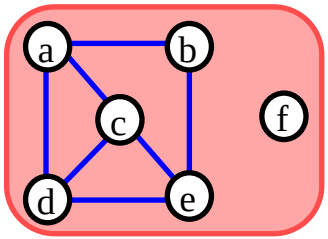


plass forbruk $O(n^2)$

```
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*

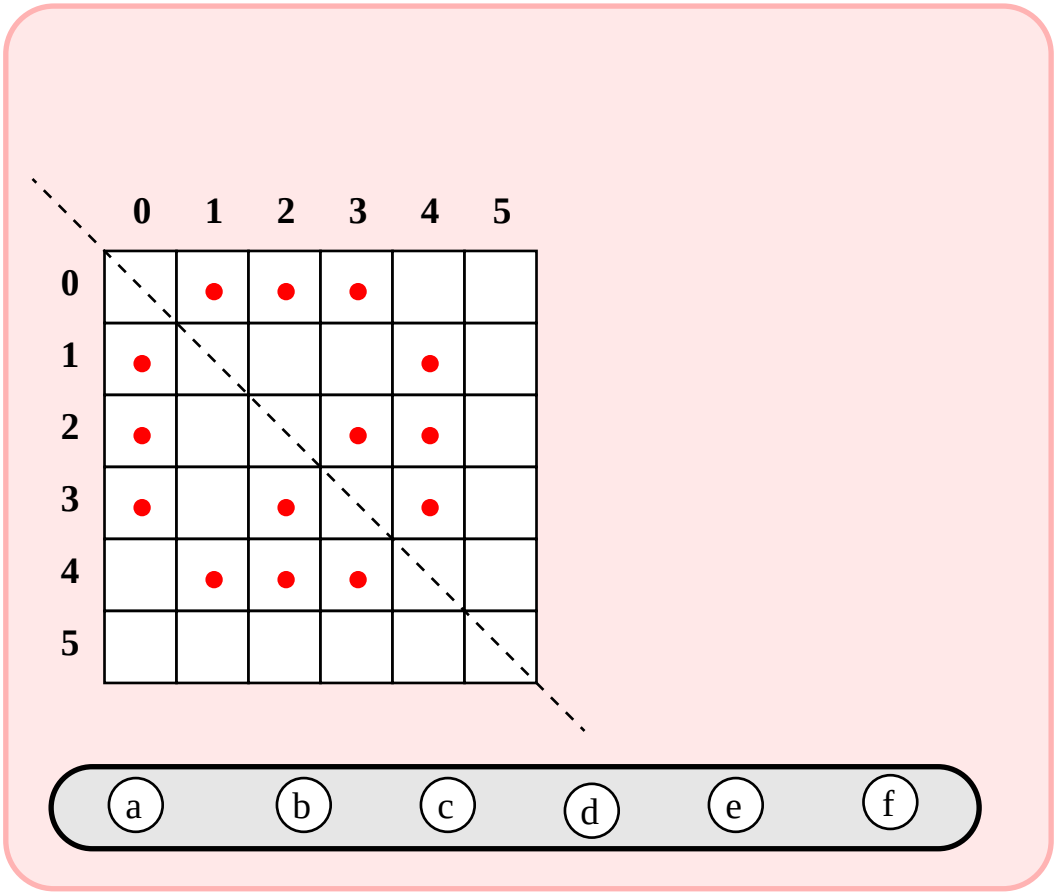


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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



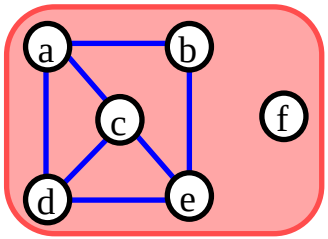
plass forbruk $O(n^2)$

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

```
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*

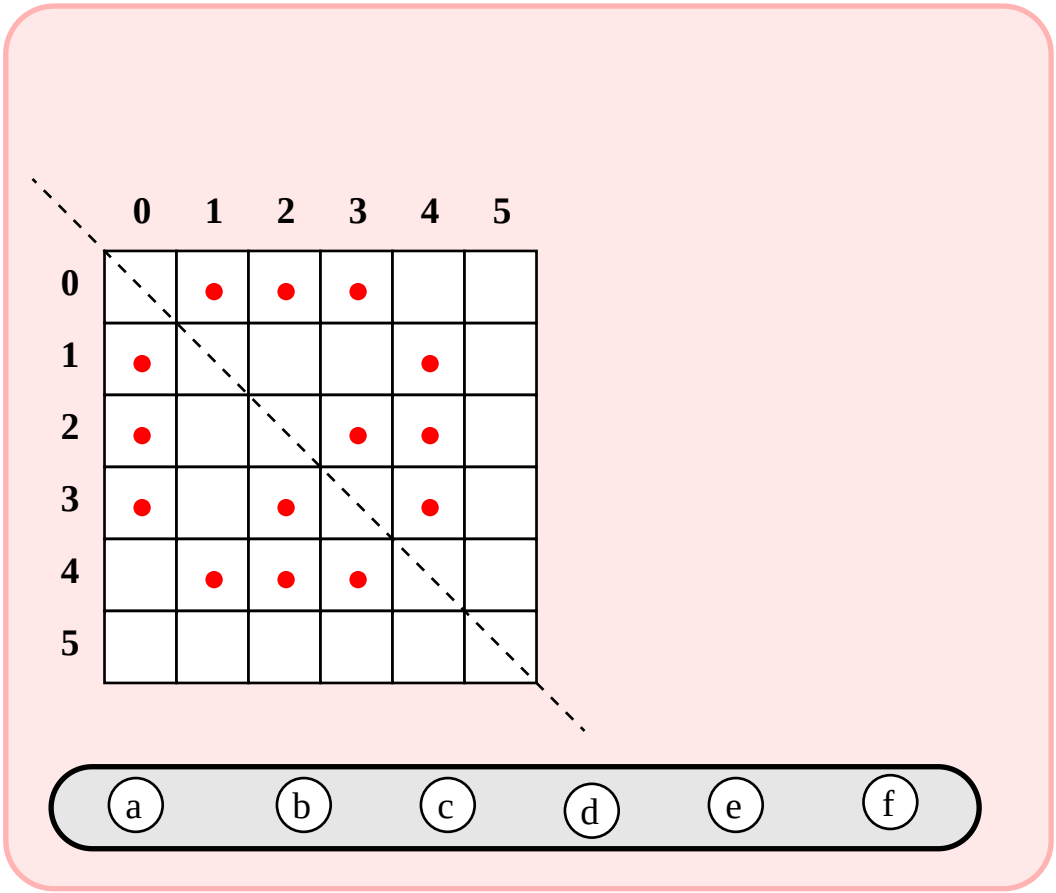


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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



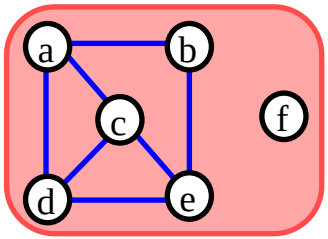
plass forbruk $O(n^2)$

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

```
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*

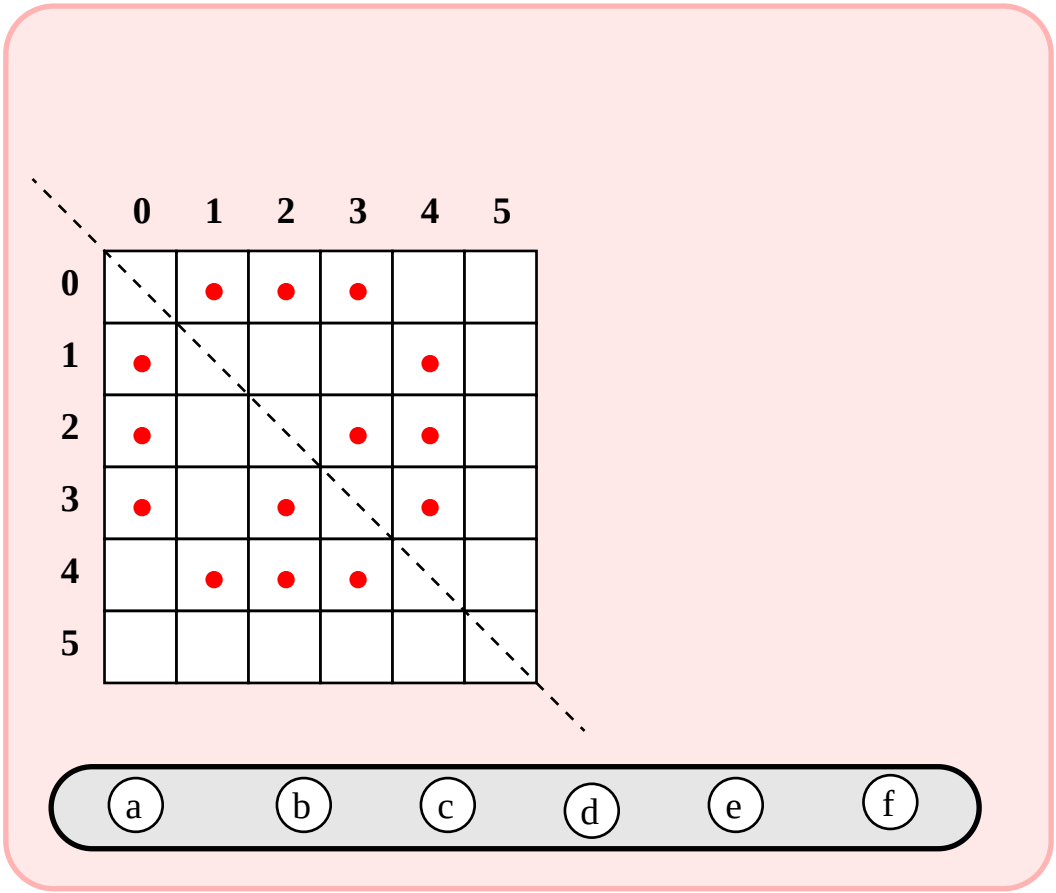


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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



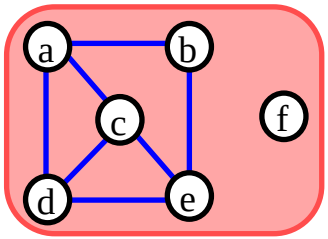
plass forbruk $O(n^2)$

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)	$O(\deg v \mid u)$	$O(1)$

```
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*

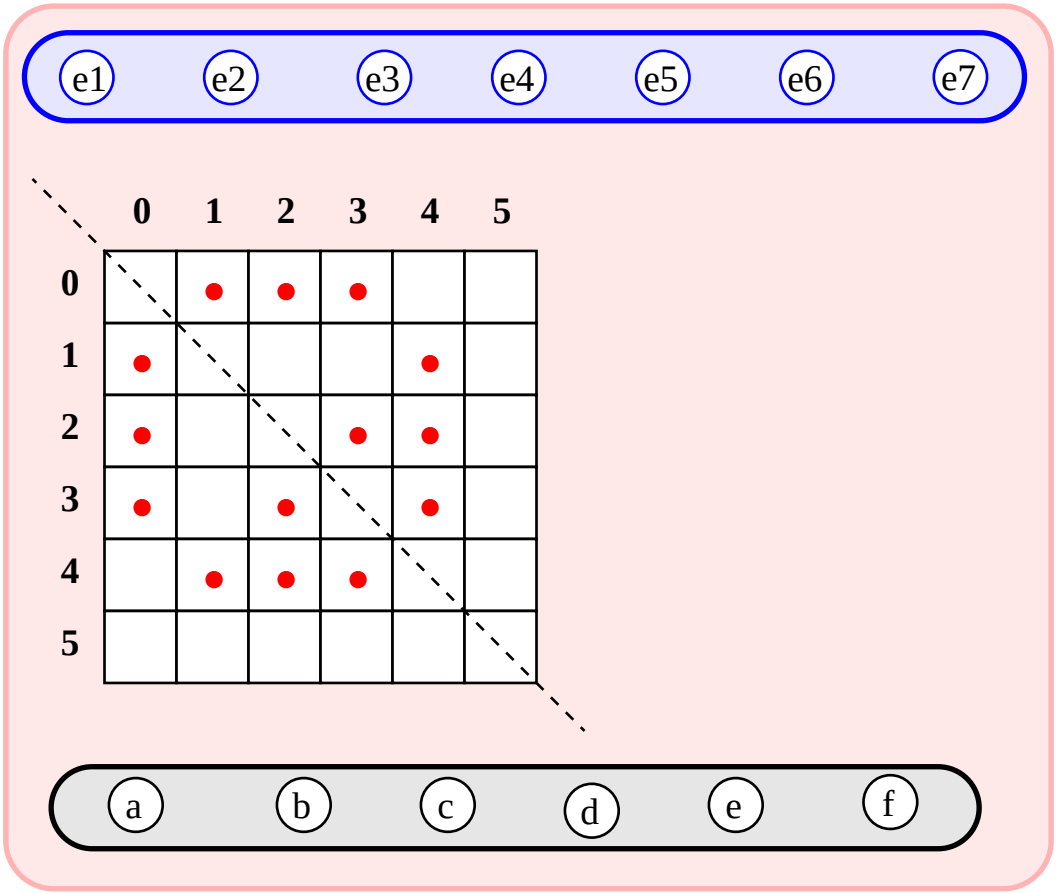


Enumerer alle noder fra 0...n-1:

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

i en **n** x**n** 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



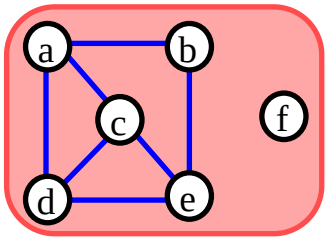
plass forbruk $O(n^2)$

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)	$O(\deg v \mid u)$	$O(1)$

```
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*

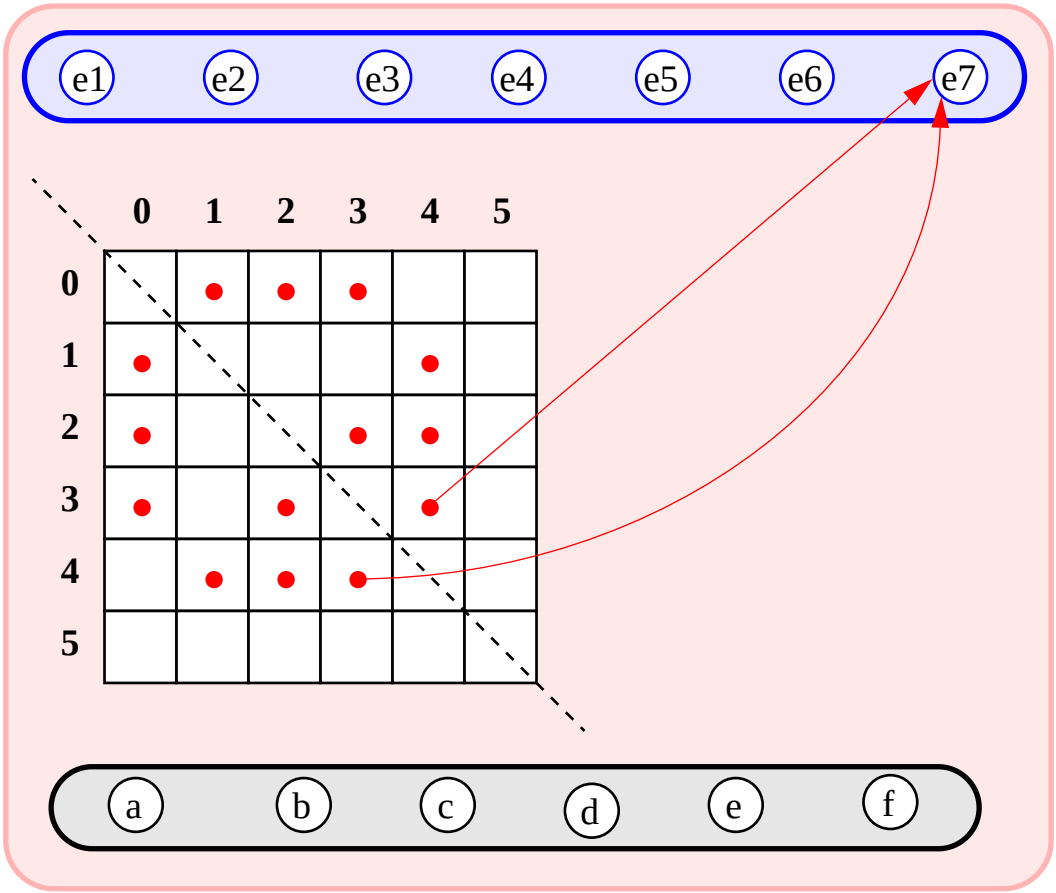


Enumerer alle noder fra 0...n-1:

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

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

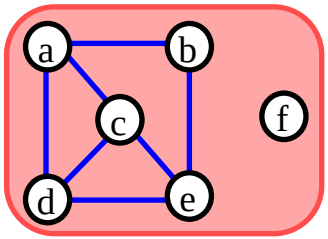
- **A[i][j]** == **e** (**true**) hviss G har en kant e=(i,j)
- **A[i][j]** == **null** (**false**) hviss (i,j) ikke er en kant i G



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)	$O(\deg v \mid u)$	$O(1)$

3. Implementasjon av Graph *med Nabo-Matrise*

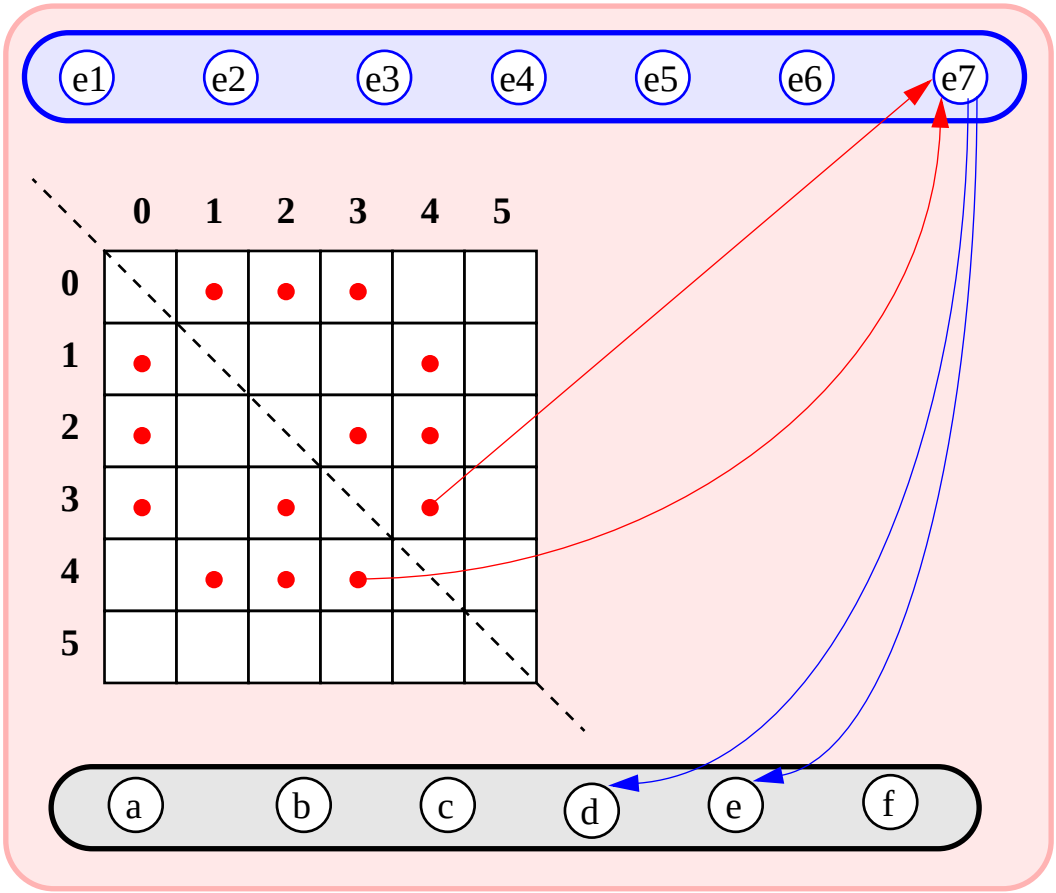


Enumerer alle noder fra 0...n-1:

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

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

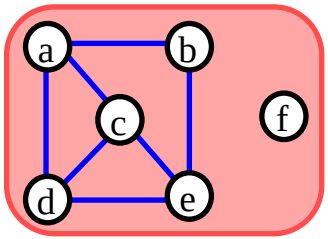
- **A[i][j]** == **e** (**true**) hviss G har en kant e=(i,j)
- **A[i][j]** == **null** (**false**) hviss (i,j) ikke er en kant i G



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

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

3. Implementasjon av Graph *med Nabo-Matrise*

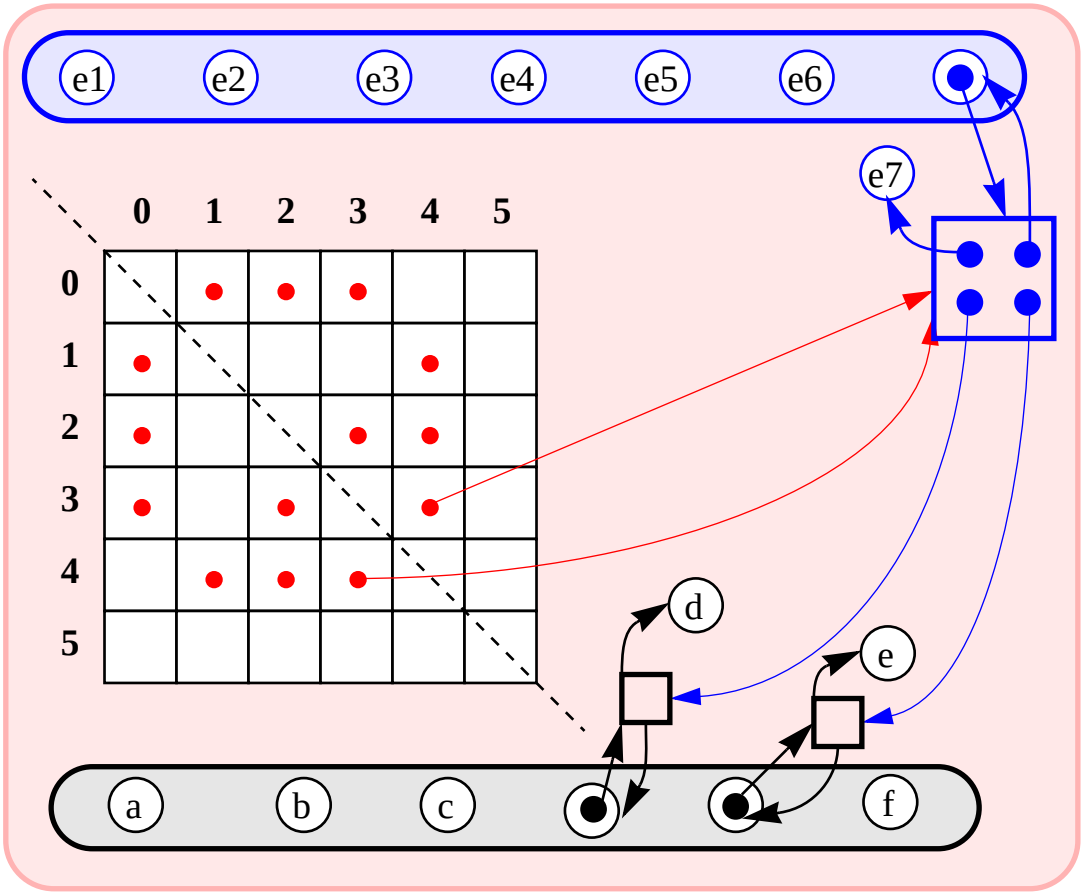


Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
|	|	|	|	|	|	no
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 (i,j) ikke er en kant i G

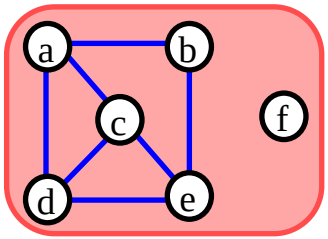


plass forbruk $O(\mathbf{n^2})$

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

```
Edge insertEdge(Vertex v,u, Object o) {
  if (ok(v) && ok(u) && !areAdjacent(v,u))
  { NMEdge e = new NMEdge(v,u,o);
    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(“”);  }
```

3. Implementasjon av Graph *med Nabo-Matrise*

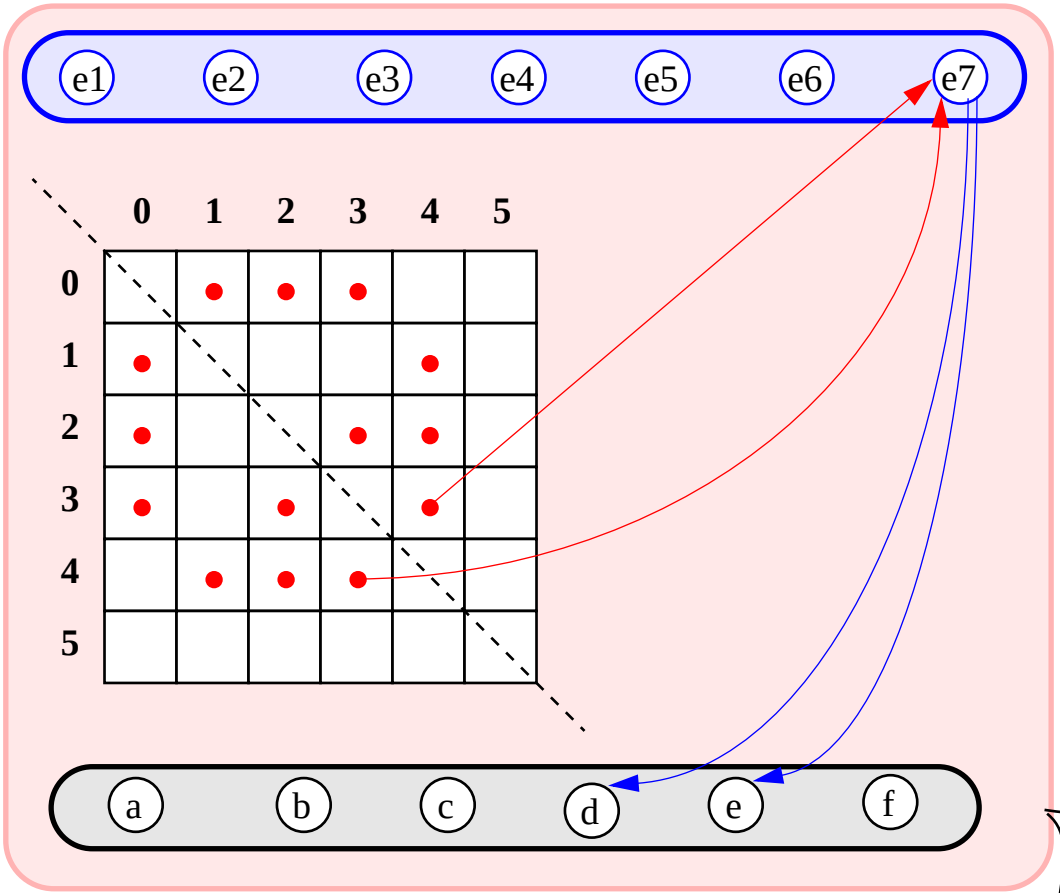


Enumerer alle noder fra 0...n-1:

a	b	c	d	e	f	
|	|	|	|	|	|	no
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 (i,j) ikke er en kant i G



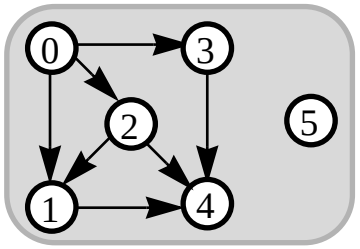
plass forbruk $O(n^2)$

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)	$O(\deg v \mid u)$	$O(1)$

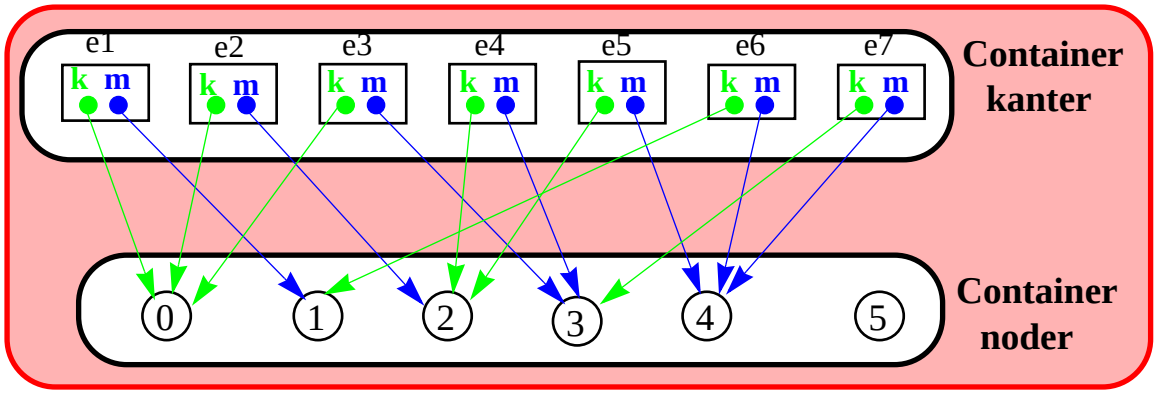
```
Edge insertEdge(Vertex v,u, Object o) {
  if (ok(v) && ok(u) && !areAdjacent(v,u))
  { NMEdge e = new NMEdge(v,u,o,this);
    e . setPos ( kanter . insertLast (e) );
    ((KLNode) v).inc(); ((KLNode) u).inc();
    A[no(v)] [no(u)]  = e ;
    return e;
  } else if (areAdjacent(v,u))  throw ...
  else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

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

4. Implementasjon av rettet Graph



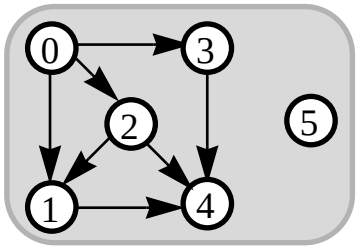
Kant-Liste



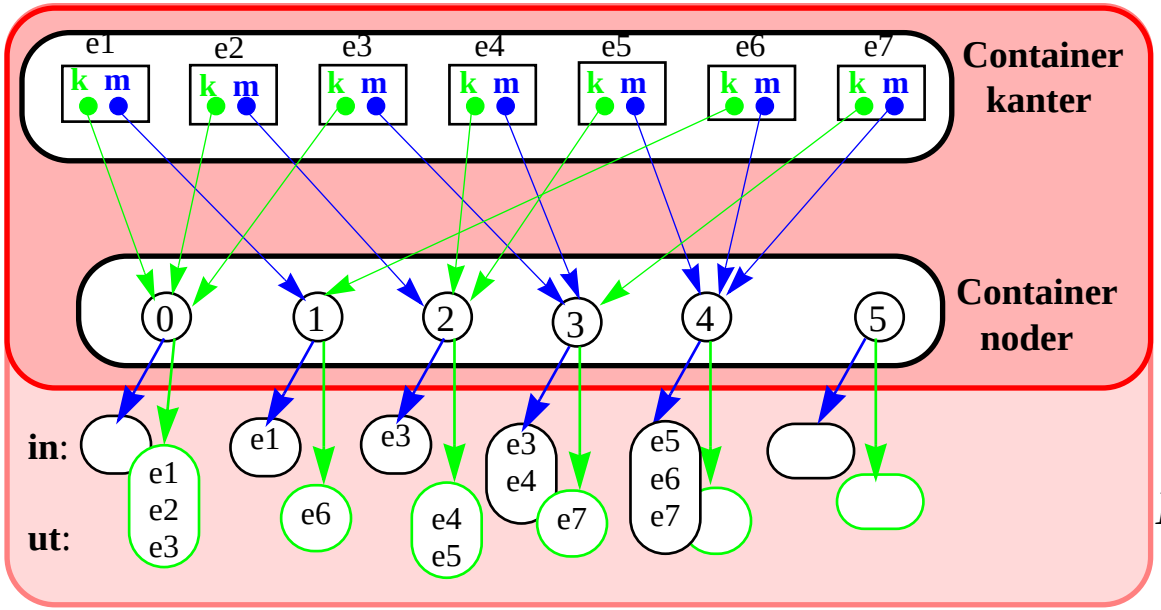
urettet implementasjon trenger ikke

- *å skille mellom* **Vertex origin** *og* **Vertex destination** *i Kant-klassen,*

4. Implementasjon av rettet Graph



Kant-Liste

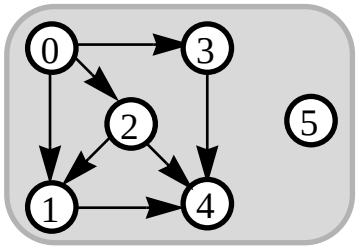


Nabo-Liste

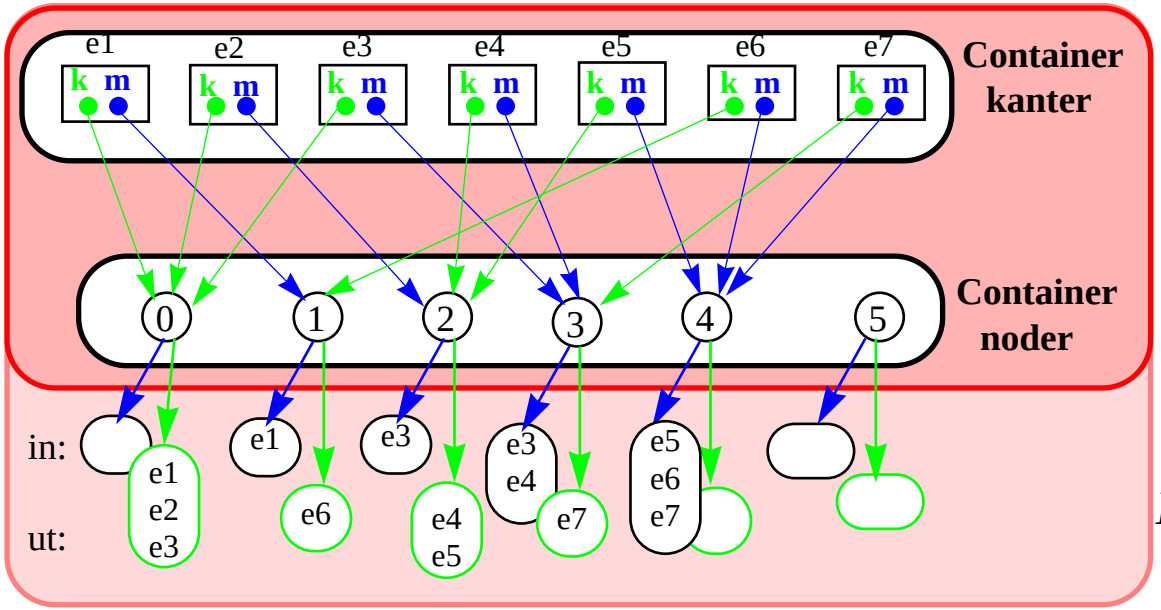
urettet implementasjon trenger ikke

- *å skille mellom* **Vertex origin** *og* **Vertex destination** *i Kant-klassen, eller*
- *mellom* **in- og ut-** *samlinger i Node-klassen*

4. Implementasjon av rettet Graph



Kant-Liste

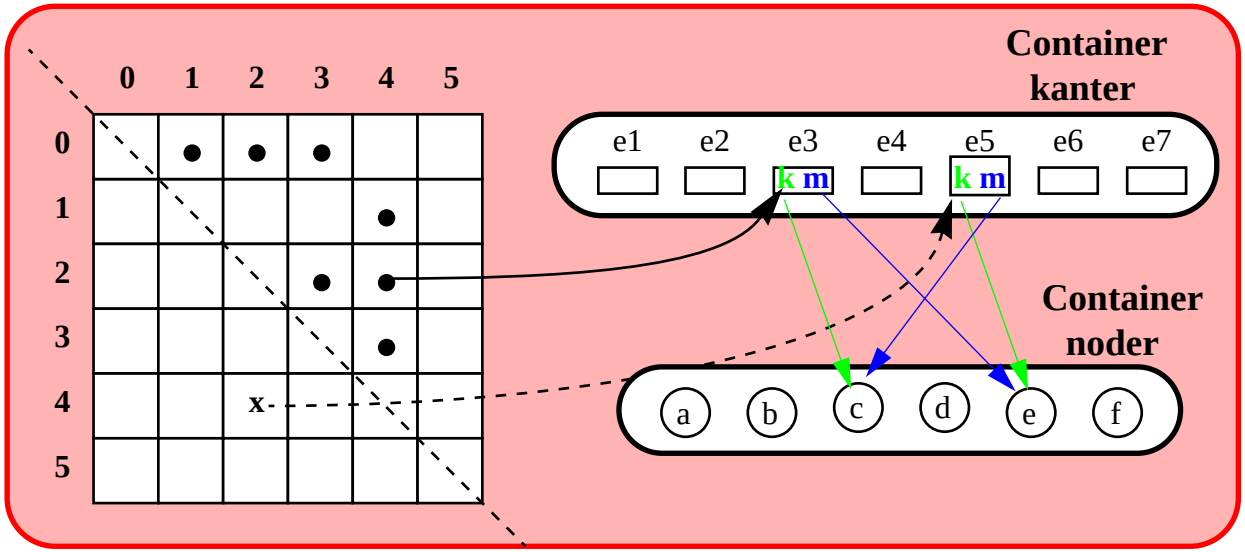


urettet implementasjon trenger ikke

- *å skille mellom* **Vertex origin** *og* **Vertex destination** *i Kant-klassen, eller*
- *mellom* **in- og ut-** *samlinger i Node-klassen*

Nabo-Liste

Nabo-Matrise



Implementasjoner av Graph

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

kompleksitet operasjon		Kant-Liste	Nabo-Liste	Nabo-Matrise
	numVertices(), numEdges()	1	1	1
vertices() / edges()	un/directedEdges()	n / k	n / k	n / k
degree(v)	in/outDegree(v)	1	1	1
endVertices(e), opposite(v,e)	origin(e), destination(e)	1	1	1
adjacentVertices(v)	in/outAdjacentVertices(v)	k	deg v	n
incidentEdges(v)	in/outIncidentEdges(v)	k	deg v	n
insertVertex(o)		1	1	n^2
removeVertex(v)		k	deg v	n^2
insertEdge(v,u,o)	inserDirectedEdge(v,u,o)	1	1	1
removeEdge(e)		1	1	1
reverseDirection(e), makeUndirected(e), ...		1	1	1
areAdjacent(v,u)		k	min(deg u,v)	1