Ordbøker

I. ORDBOK / PRIORITETSKØ = SEKVENS / LIFI

vilkårlig / minste nøkkel

vilkårlig / første Posisjon

II. IMPLEMENTASJON MED SEQUENCE

III. IMPLEMENTASJON MED HASH TABELL

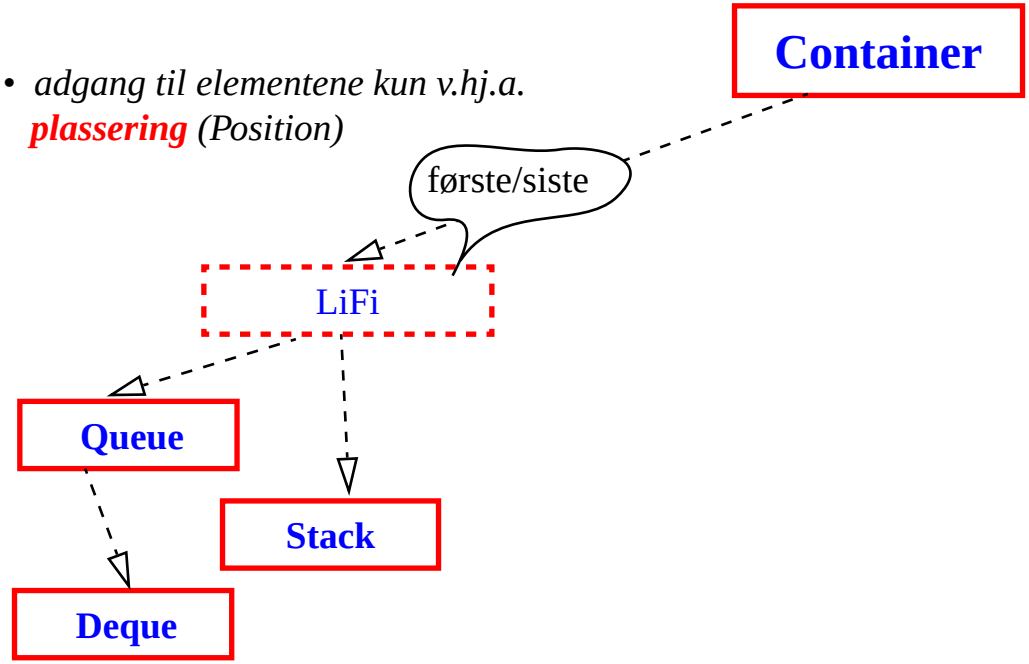
hash funksjoner

håndtering av kollisjoner

IV. IMPLEMENTASJON MED BST (BINARY SEARCH TREE)

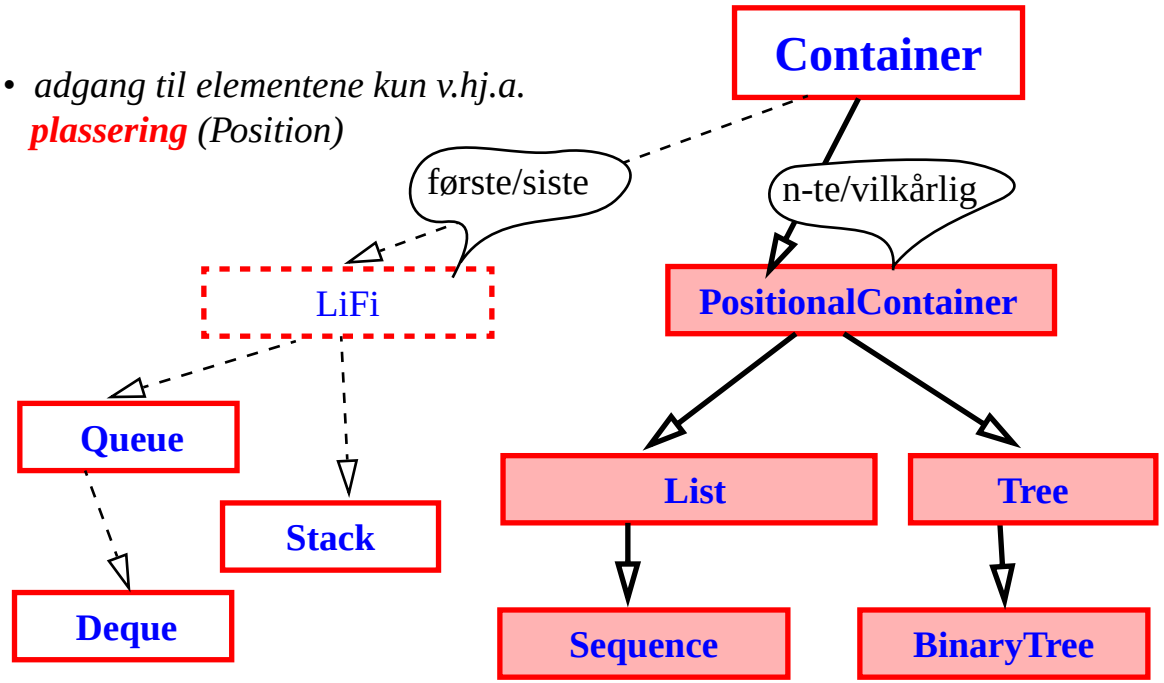
Kap. 8: (kursorisk: 8.3.3 – 8.3.7, 8.7;
unntatt: 8.6)
Kap. 9: (unntatt: 9.2.1 – 9.7)

- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*



- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*

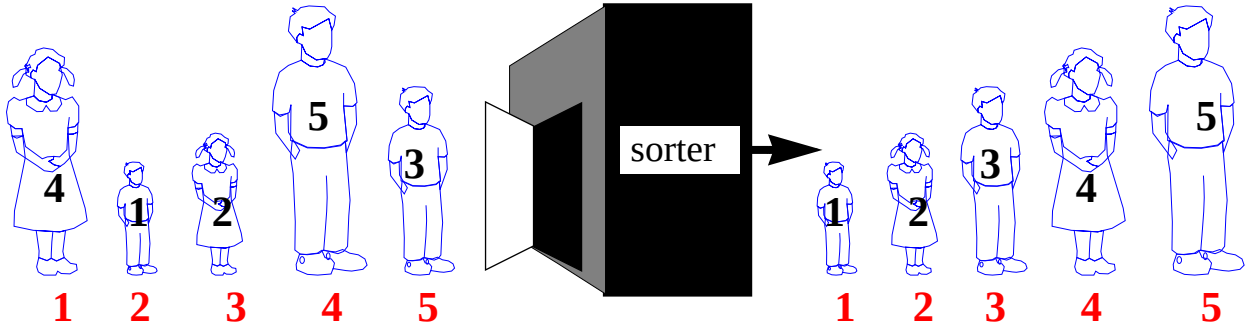


- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

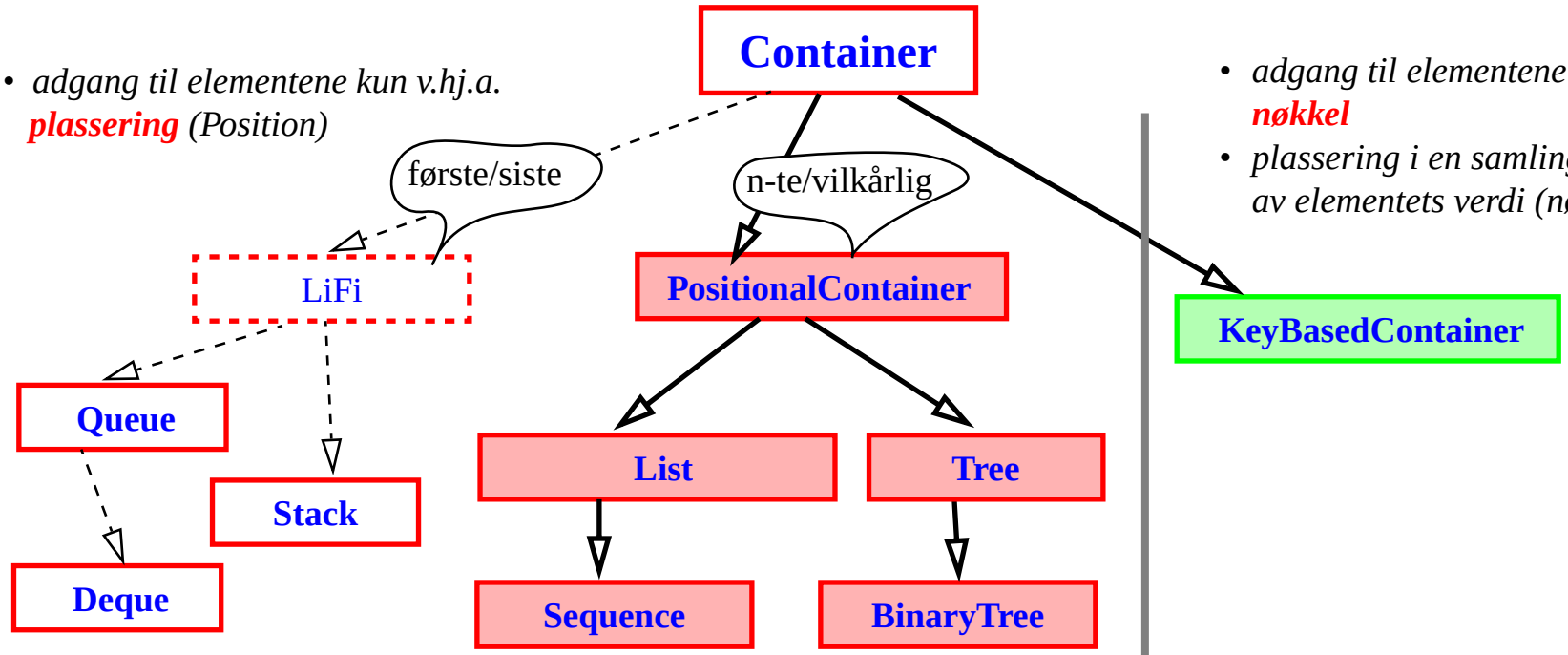
... men f.eks. Sortering er en operasjon

– på strukturer med rekkefølgen på Posisjoner, og hvor

– alle elementene kan sammenlignes (mht. en nøkkel-verdi)



- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*



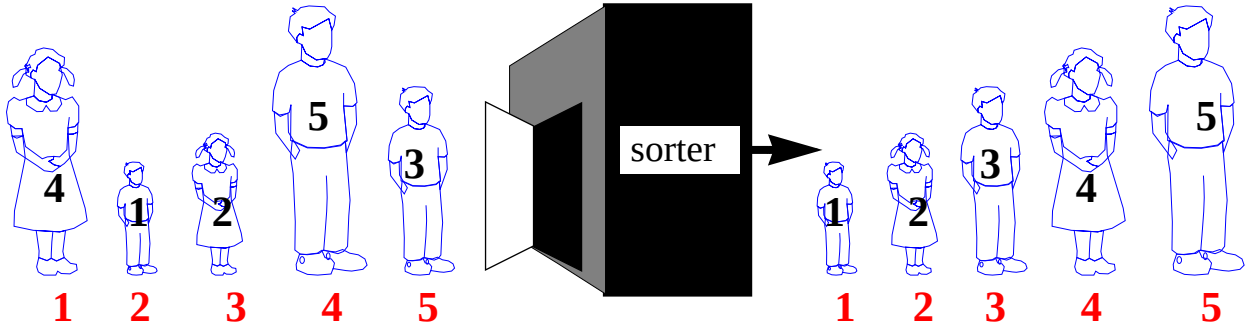
- *adgang til elementene v.hj.a.* ***nøkkel***
- *plassering i en samling* ***avhenger*** *av elementets verdi (nøkkel)*

- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

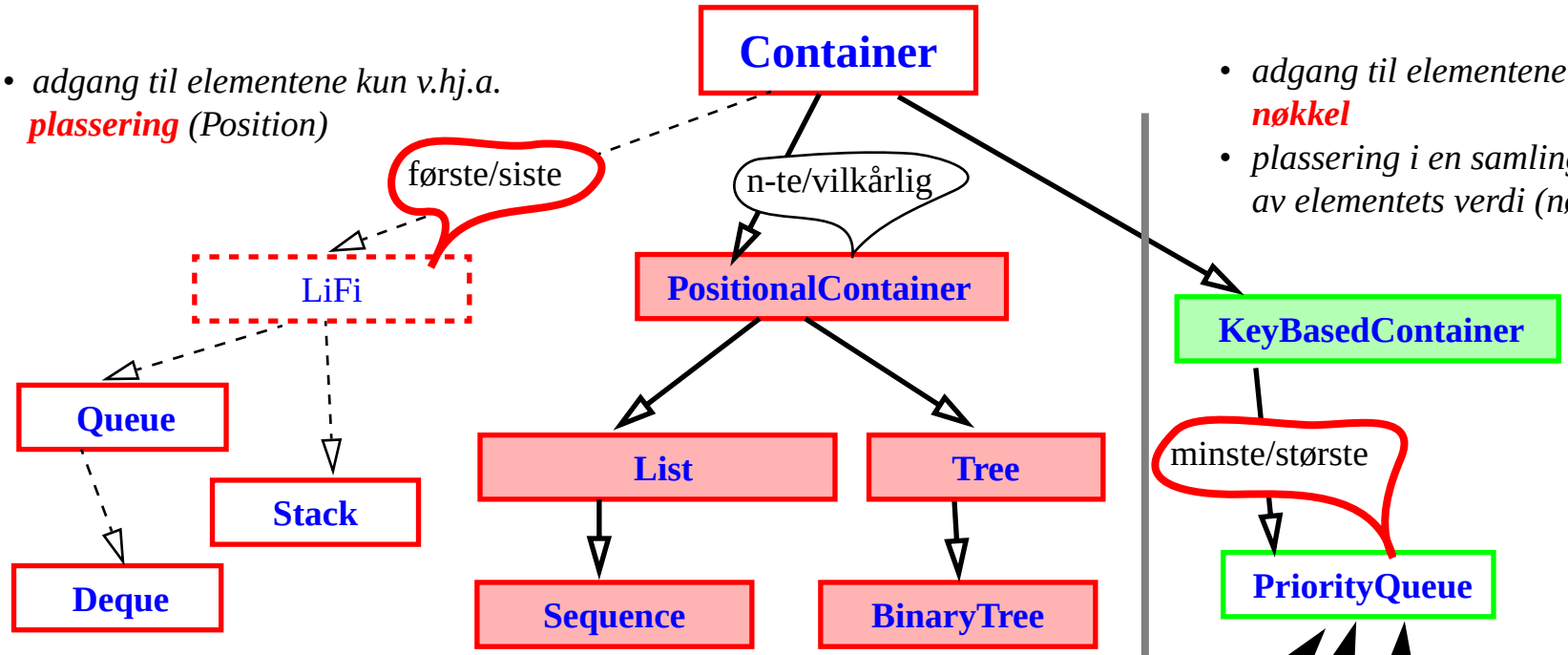
... men f.eks. Sortering er en operasjon

– på strukturer med rekkefølgen på Posisjoner, og hvor

– alle elementene kan sammenlignes (mht. en nøkkel-verdi)



- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*



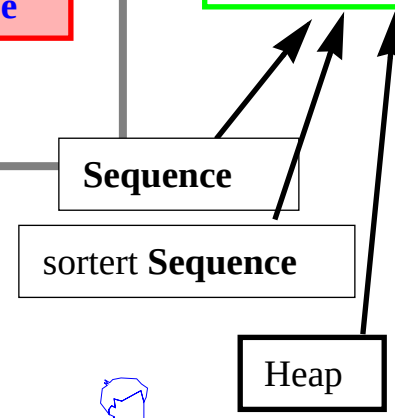
- *adgang til elementene v.hj.a.* ***nøkkel***
- *plassering i en samling* ***avhenger*** *av elementets verdi (nøkkel)*

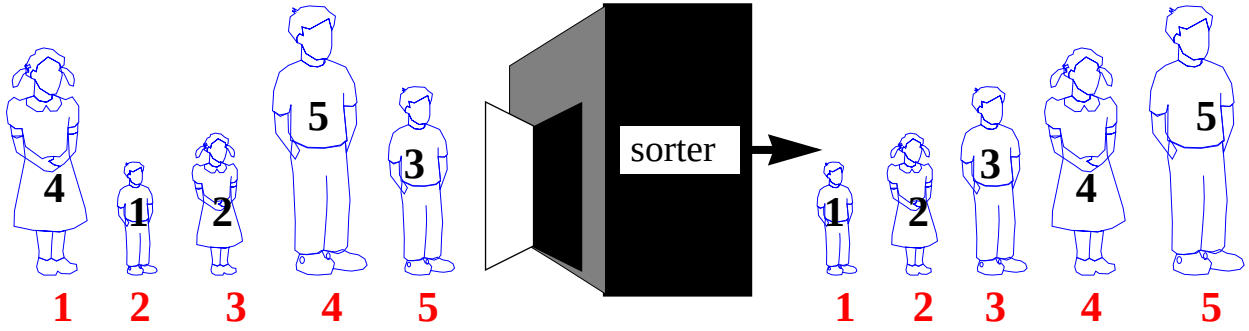
- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

... men f.eks. Sortering er en operasjon

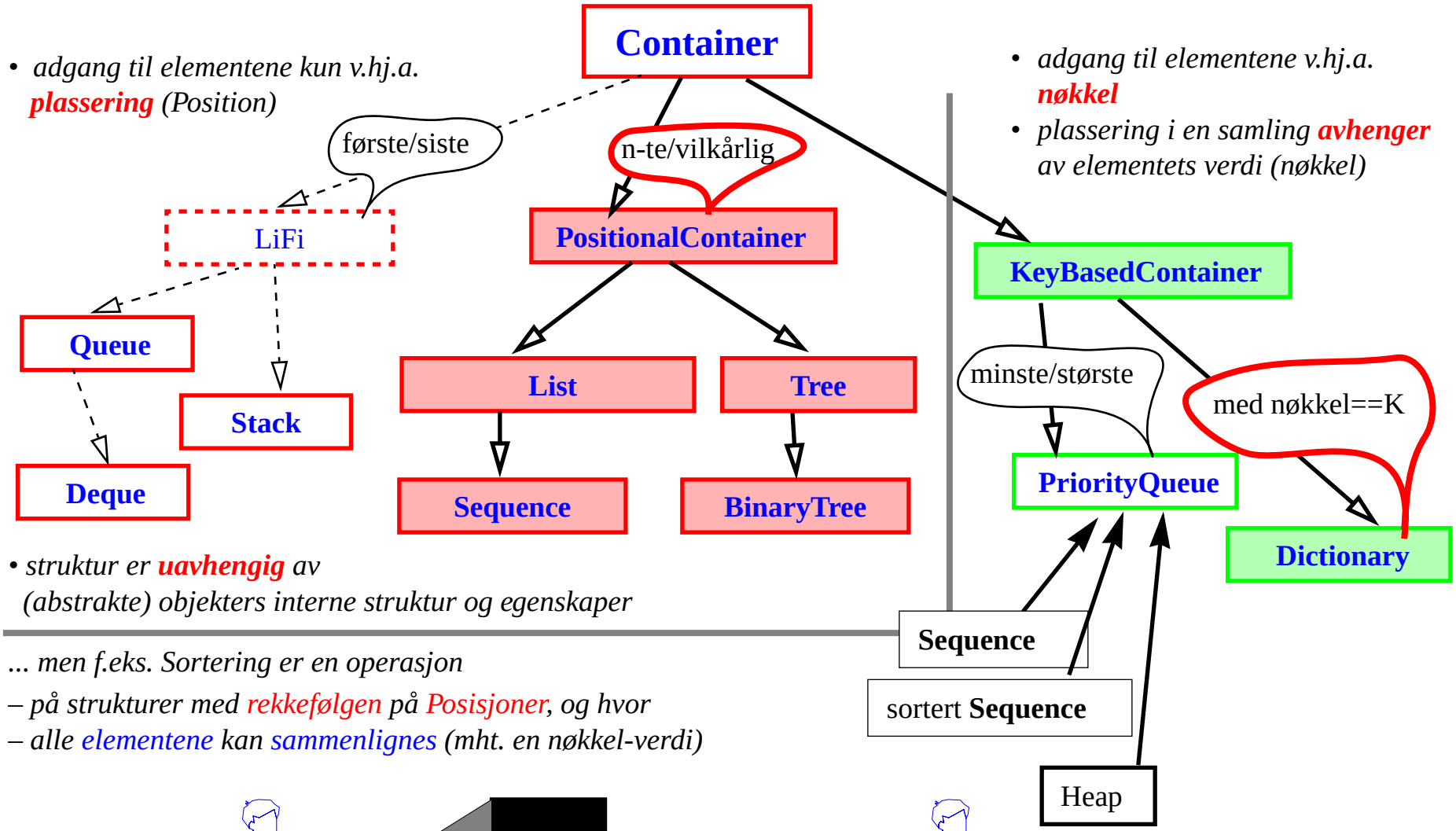
– på strukturer med rekkefølgen på Posisjoner, og hvor

– alle elementene kan sammenlignes (mht. en nøkkel-verdi)





- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*



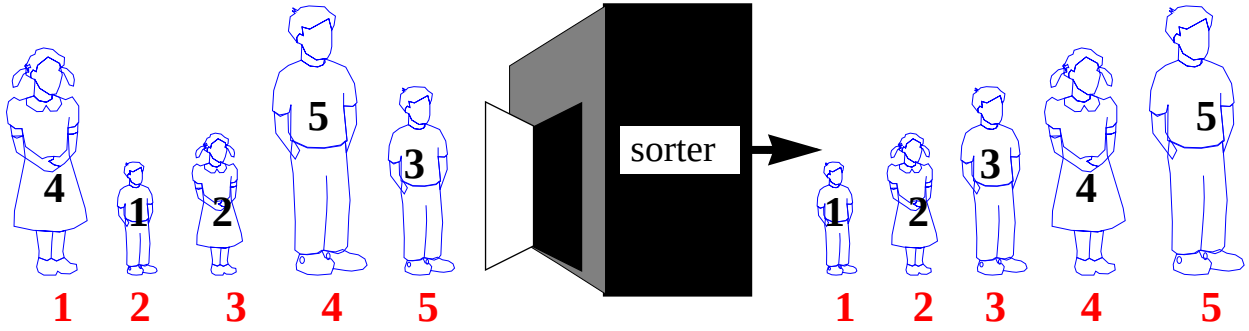
- *adgang til elementene v.hj.a.* ***nøkkel***
- *plassering i en samling* ***avhenger*** *av elementets verdi (nøkkel)*

- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

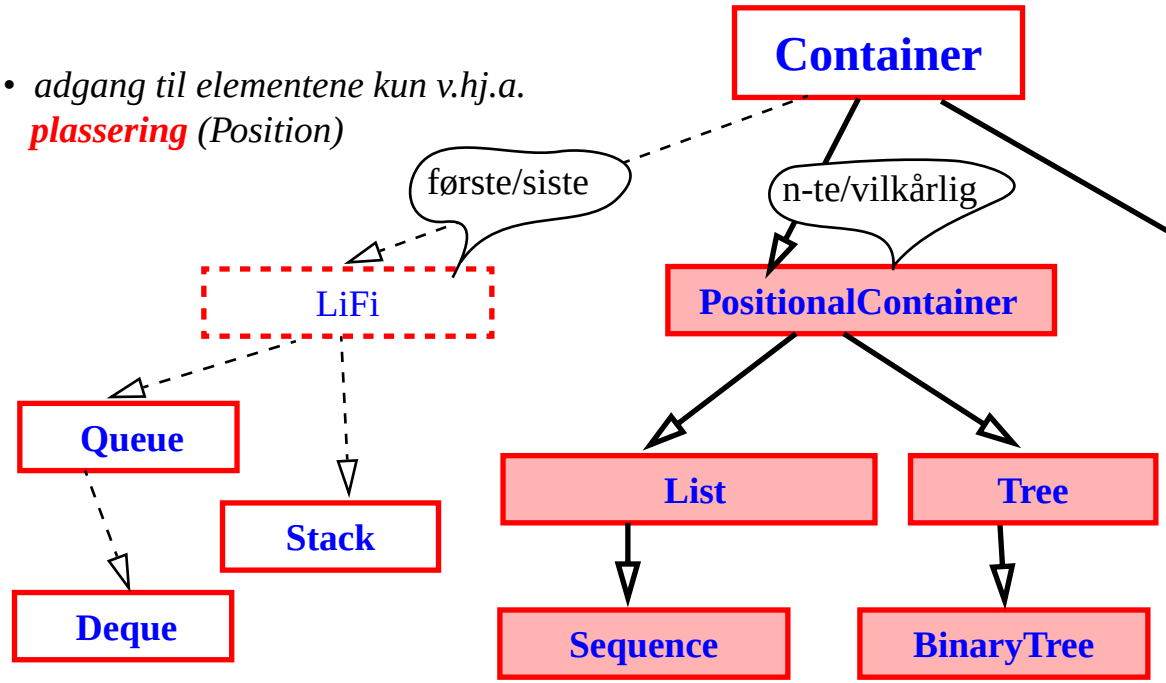
... men f.eks. Sortering er en operasjon

– på strukturer med rekkefølgen på Posisjoner, og hvor

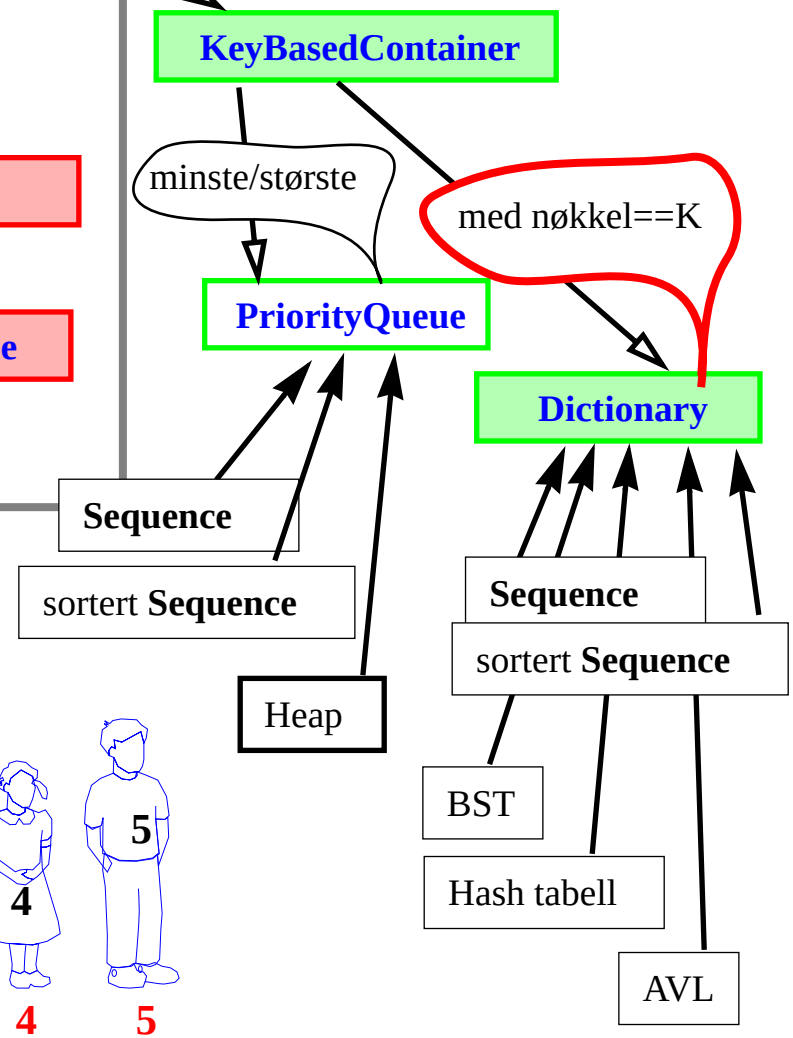
– alle elementene kan sammenlignes (mht. en nøkkel-verdi)



- *adgang til elementene kun v.hj.a.* ***plassering*** *(Position)*



- *adgang til elementene v.hj.a.* ***nøkkel***
- *plassering i en samling* ***avhenger*** *av elementets verdi (nøkkel)*

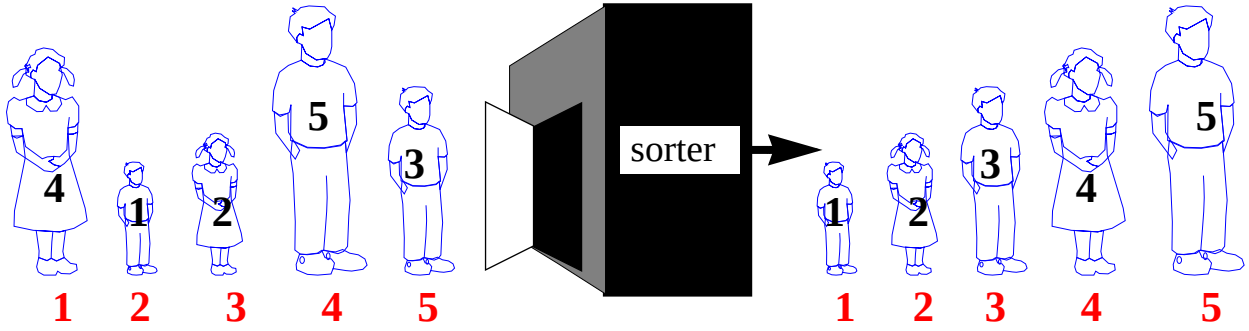


- *struktur er* ***uavhengig*** *av (abstrakte) objekters interne struktur og egenskaper*

... men f.eks. Sortering er en operasjon

– på strukturer med rekkefølgen på Posisjoner, og hvor

– alle elementene kan sammenlignes (mht. en nøkkel-verdi)



En Ordbok

er en funksjon

f: *Nøkler*	$\rightarrow$	*DataObjekter*
personnummer	$\rightarrow$	Person
fødselsdato + kjønn (+???)	$\rightarrow$	personnummer
navn + adresse	$\rightarrow$	telefonnummer
navn	$\rightarrow$	telefonnummer
studentnavn (+???)	$\rightarrow$	studentID
studentID	$\rightarrow$	karakterutskrift
x	$\rightarrow$	x^2

En Ordbok (tabell, symboltabell, “map”...)

er en funksjon

f: *Nøkler*	→	*DataObjekter*
personnummer	→	Person
fødselsdato + kjønn (+???)	→	personnummer
navn + adresse	→	telefonnummer
navn	→	telefonnummer
studentnavn (+???)	→	studentID
studentID	→	karakterutskrift
x	→	x^2

En Ordbok (tabell, symboltabell, “map”...)

er en funksjon

f: *Nøkler*	→	*DataObjekter*
personnummer	→	Person
fødselsdato + kjønn (+???)	→	personnummer
navn + adresse	→	telefonnummer
navn	→	telefonnummer
studentnavn (+???)	→	studentID
studentID	→	karakterutskrift
x	→	x^2

som tillatter en

- å sette inn nye nøkkel-objekt par
- finne et objekt med en gitt nøkkel
- (finne alle objektene med en gitt nøkkel)
- fjerne et objekt med en gitt nøkkel
- finne neste/forrige objektet (mht. nøkkel)

En Ordbok (tabell, symboltabell, “map”...)

er en funksjon

f: *Nøkler*	→	*DataObjekter*
personnummer	→	Person
fødselsdato + kjønn (+???)	→	personnummer
navn + adresse	→	telefonnummer
navn	→	telefonnummer
studentnavn (+???)	→	studentID
studentID	→	karakterutskrift
x	→	x^2

som tillatter en

- å sette inn nye nøkkel-objekt par
- finne et objekt med en gitt nøkkel
- (finne alle objektene med en gitt nøkkel)
- fjerne et objekt med en gitt nøkkel
- finne neste/forrige objektet (mht. nøkkel)

```
public interface DictionaryWithoutLocators extends KeyBasedContainer {
            void  insert        (Object key, Object o)
          Object  find          (Object key)
   ObjectIterator findAll       (Object key)
          Object  remove        (Object key)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

}
```

En Ordbok (tabell, symboltabell, “map”...)

er en funksjon

f: *Nøkler*	→	*DataObjekter*
personnummer	→	Person
fødselsdato + kjønn (+???)	→	personnummer
navn + adresse	→	telefonnummer
navn	→	telefonnummer
studentnavn (+???)	→	studentID
studentID	→	karakterutskrift
x	→	x^2

som tillatter en

- å sette inn nye nøkkel-objekt par
- finne et objekt med en gitt nøkkel
- (finne alle objektene med en gitt nøkkel)
- fjerne et objekt med en gitt nøkkel
- finne neste/forrige objektet (mht. nøkkel)

```
public interface DictionaryWithoutLocators extends KeyBasedContainer {
            void  insert            (Object key, Object o)
          Object  find              (Object key)
   ObjectIterator findAll           (Object key)
          Object  remove            (Object key)
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          Object  closestElemAfter  (Object key)
          Object  closestElemBefore (Object key)
          Object  closestKeyAfter   (Object key)
          Object  closestKeyBefore  (Object key)
}
```

En Ordbok med Locator

```
public interface Locator {
    Object element()
    Object key()
```

er en funksjon

f: *Nøkler*	→	*Locator for DataObjekter*
personnummer	→	Person
fødselsdato + kjønn (+???)	→	personnummer
navn + adresse	→	telefonnummer
navn	→	telefonnummer
studentnavn (+???)	→	studentID
studentID	→	karakterutskrift
x	→	x^2

som tillatter en

- å sette inn nye nøkkel-objekt par
- finne et objekt med en gitt nøkkel
- (finne alle objektene med en gitt nøkkel)
- fjerne et objekt med en gitt nøkkel
- finne neste/forrige objektet (mht. nøkkel)

```
public interface (Ordered)Dictionary extends KeyBasedContainer {
        Locator  insert          (Object key, Object o)
        Locator  find            (Object key)
  LocatorIterator findAll        (Object key)
           void  remove          (Locator loc)
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        Locator  after           (Locator loc)
        Locator  before          (Locator loc)
        Locator  closestAfter    (Object key)
        Locator  closestBefore   (Object key)
}
```

Dictionary og OrderedDictionary

package jdsl.core.api

```
public interface Locator {
    Object element()
    Object key()
```

```
public interface KeyBasedContainer extends Container {
    LocatorIterator locators()
    ObjectIterator keys()
/** Inserts a <key, element> pair */
    Locator insert(Object k, Object o)
/** Removes an element from this Container. */
    void remove(Locator)
/** Inserts a Locator into this Container. */
    void insert(Locator)
    Object replaceElement(Locator l, Object o)
/** Changes the key of Locator's element.
    @return the old key of this Locator's element */
    Object replaceKey(Locator, Object)
}
```

```
public interface Dictionary extends KeyBasedContainer {
/** @param key to search for an object
    @returns Locator mapped to key – NO_SUCH_KEY if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    Locator find(Object key)
/** @param key to search for an object
    @returns all Locators mapped to key – empty if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    LocatorIterator findAll(Object key)
/** Special value (rather than exception) returned when no object
    with a specified key was found */
    Locator NO_SUCH_KEY;
```

```
public interface OrderedDictionary extends Dictionary {
/** @returns Locator sequentially after/before loc – BOUNDARY_VIOLATION if goes out of scope
    @exception InvalidLocatorExc if loc is invalid (e.g. not within this Container) */
    Locator after/before(Locator loc)
/** @returns a Locator whose key is equal to or just after/before key – BOUNDARY_VIOLATION if no such exists
    @exception InvalidKeyExc if key is invalid for this Container */
    Locator closestAfter/Before(Object key)
/** Special Locator (rather than exception) returned when a requested Locator was not found */
    Locator BOUNDARY_VIOLATION;
```

Dictionary og OrderedDictionary

package jdsl.core.api

```
public interface Locator {
    Object element()
    Object key()
```

```
public interface KeyBasedContainer extends Container {
    LocatorIterator locators()
    ObjectIterator keys()
/** Inserts a <key, element> pair */
    Locator insert(Object k, Object o)
/** Removes an element from this Container. */
    void remove(Locator)
/** Inserts a Locator into this Container. */
    void insert(Locator)
    Object replaceElement(Locator l, Object o)
/** Changes the key of Locator's element.
    @return the old key of this Locator's element */
    Object replaceKey(Locator, Object)
}
```

```
public interface Dictionary extends KeyBasedContainer {
/** @param key to search for an object
    @returns Locator mapped to key – NO_SUCH_KEY if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    Locator find(Object key)
/** @param key to search for an object
    @returns all Locators mapped to key – empty if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    LocatorIterator findAll(Object key)
/** Special value (rather than exception) returned when no object
    with a specified key was found */
    Locator NO_SUCH_KEY;
```

```
public interface OrderedDictionary extends Dictionary {
/** @returns Locator sequentially after/before loc – BOUNDARY_VIOLATION if goes out of scope
    @exception InvalidLocatorExc if loc is invalid (e.g. not within this Container) */
    Locator after/before(Locator loc)
/** @returns a Locator whose key is equal to or just after/before key – BOUNDARY_VIOLATION if no such exists
    @exception InvalidKeyExc if key is invalid for this Container */
    Locator closestAfter/Before(Object key)
/** Special Locator (rather than exception) returned when a requested Locator was not found */
    Locator BOUNDARY_VIOLATION;
```

Dictionary og OrderedDictionary

package jdsl.core.api

```
public interface Locator {
    Object element()
    Object key()
```

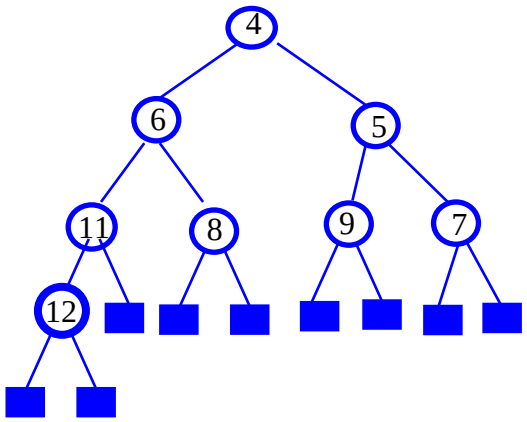
```
public interface KeyBasedContainer extends Container {
    LocatorIterator locators()
    ObjectIterator keys()
/** Inserts a <key, element> pair */
    Locator insert(Object k, Object o)
/** Removes an element from this Container. */
    void remove(Locator)
/** Inserts a Locator into this Container. */
    void insert(Locator)
    Object replaceElement(Locator l, Object o)
/** Changes the key of Locator's element.
    @return the old key of this Locator's element */
    Object replaceKey(Locator, Object)
}
```

```
public interface Dictionary extends KeyBasedContainer {
/** @param key to search for an object
    @returns Locator mapped to key – NO_SUCH_KEY if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    Locator find(Object key)
/** @param key to search for an object
    @returns all Locators mapped to key – empty if not found
    @exception InvalidKeyExc if key is not valid for this Cotainer */
    LocatorIterator findAll(Object key)
/** Special value (rather than exception) returned when no object
    with a specified key was found */
    Locator NO_SUCH_KEY;
```

```
public interface OrderedDictionary extends Dictionary {
/** @returns Locator sequentially after/before loc – BOUNDARY_VIOLATION if goes out of scope
    @exception InvalidLocatorExc if loc is invalid (e.g. not within this Container) */
    Locator after/before(Locator loc)
/** @returns a Locator whose key is equal to or just after/before key – BOUNDARY_VIOLATION if no such exists
    @exception InvalidKeyExc if key is invalid for this Container */
    Locator closestAfter/Before(Object key)
/** Special Locator (rather than exception) returned when a requested Locator was not found */
    Locator BOUNDARY_VIOLATION;
```

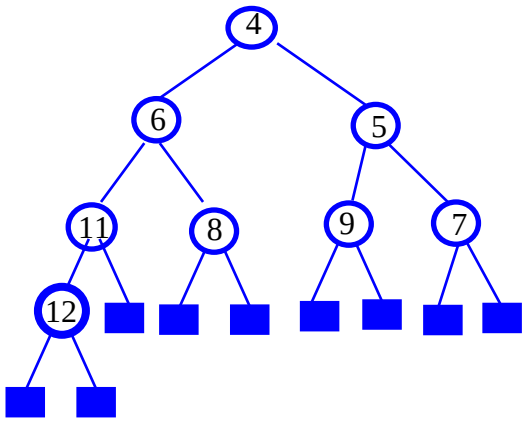
Implementasjon av Dictionary

ikke Haug



Implementasjon av Dictionary

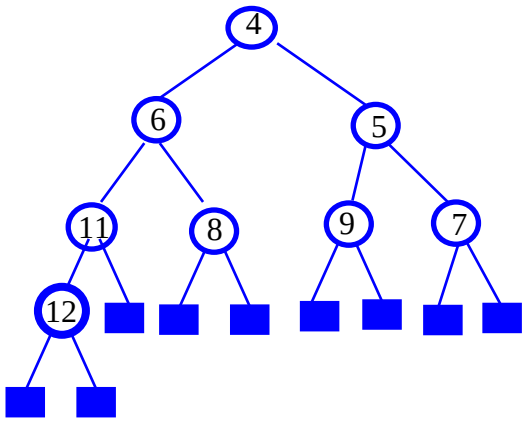
ikke Haug



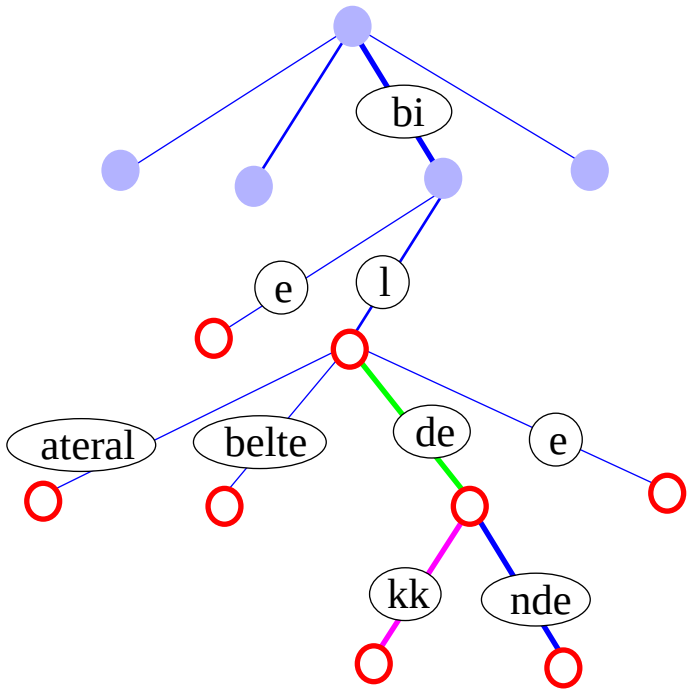
hvordan finner man ‘11’ ... ?

Implementasjon av Dictionary

ikke Haug



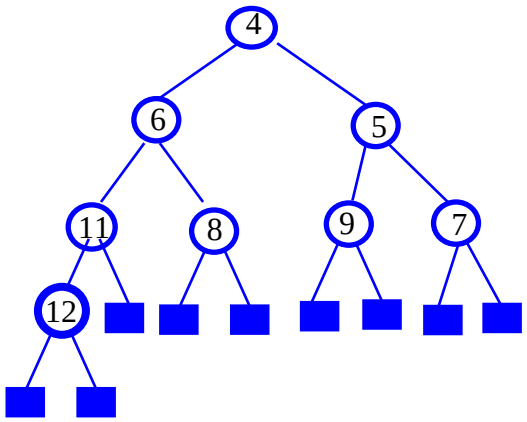
ikke Tries



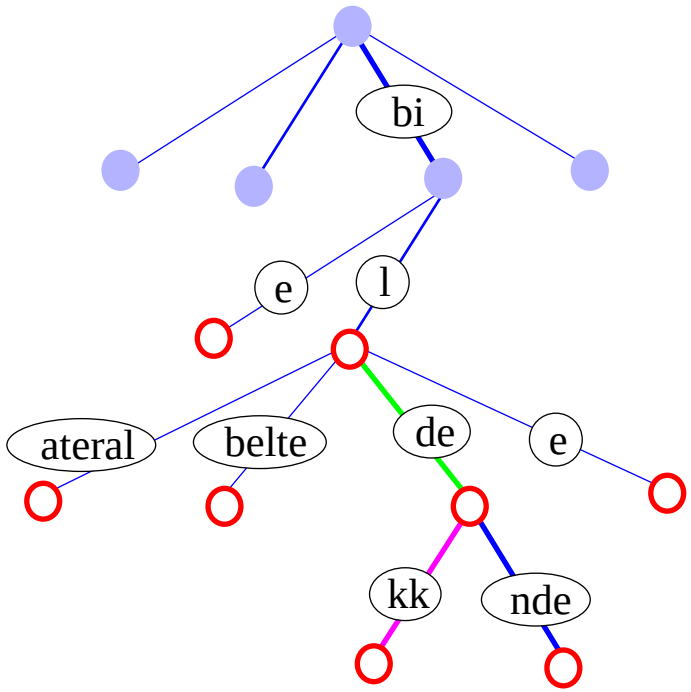
hvordan finner man ‘11’ ... ?

Implementasjon av Dictionary

ikke Haug



ikke Tries



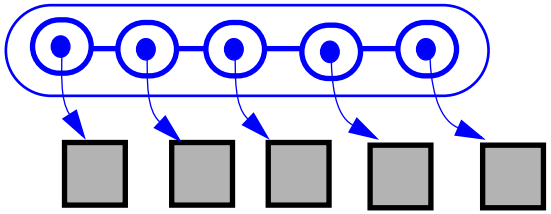
hvordan finner man ‘11’ ... ?

hva om nøkler ikke er String/Sequence ... ?

II. Implementasjon av Dictionary – *med Sequence*

I DATA REPRESENTASJON

Sequence,

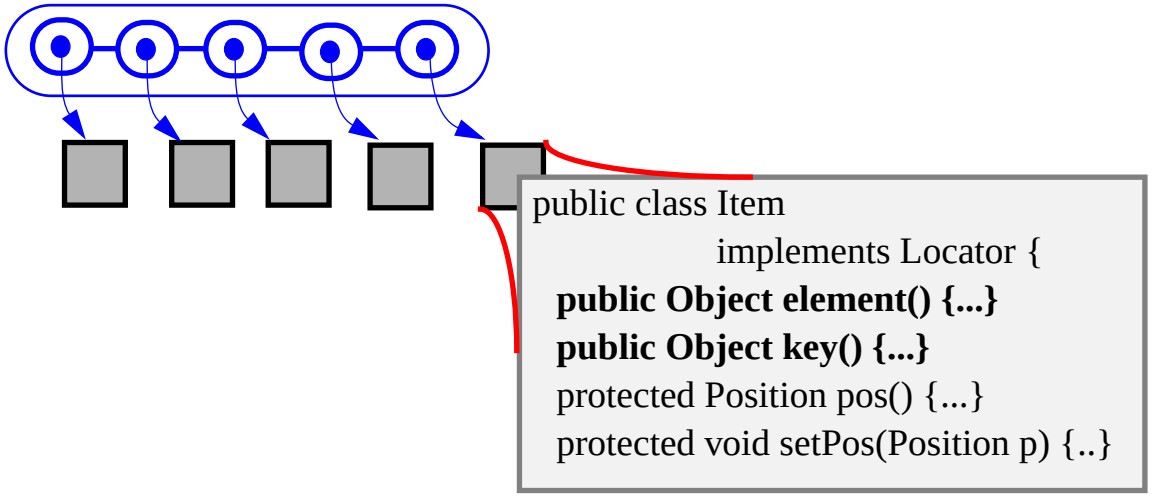


II. Implementasjon av Dictionary – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en Item

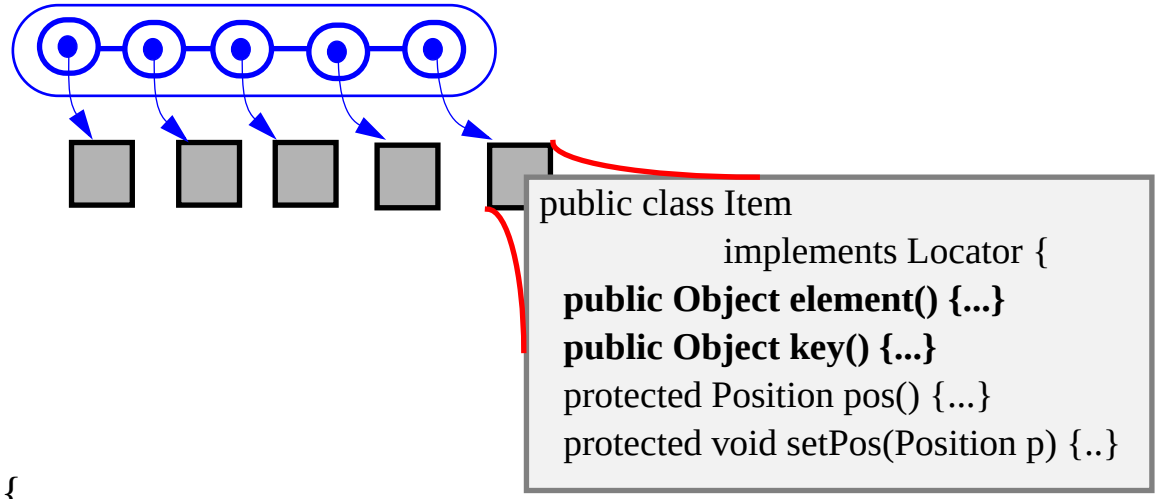


II. Implementasjon av Dictionary – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en Item



II DATA STRUKTUR

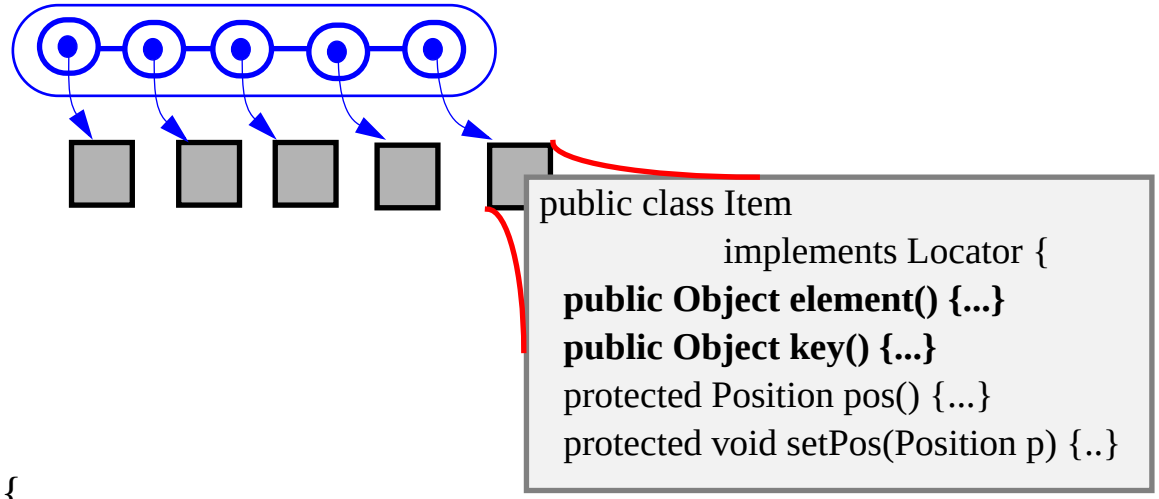
```
class DicSequence implements Dictionary {
    private Sequence S;
    private Comparator cp;
    /** @param sq bør være tom sekvens
        @param c Comparator for sammenlikning av Item med hensyn til key-objekter */
    public DicSequence(Sequence sq, Comparator c) {
      S = sq; cp = c; }
```

II. Implementasjon av Dictionary – *med Sequence*

I Data Representasjon

Sequence, der hver Position

lagrer en Item



II Data Struktur

```
class DicSequence implements Dictionary {
    private Sequence S;
    private Comparator cp;
    /** @param sq bør være tom sekvens
        @param c Comparator for sammenlikning av Item med hensyn til key-objekter */
    public DicSequence(Sequence sq, Comparator c) {
      S = sq; cp = c; }
```

III Data Invariant

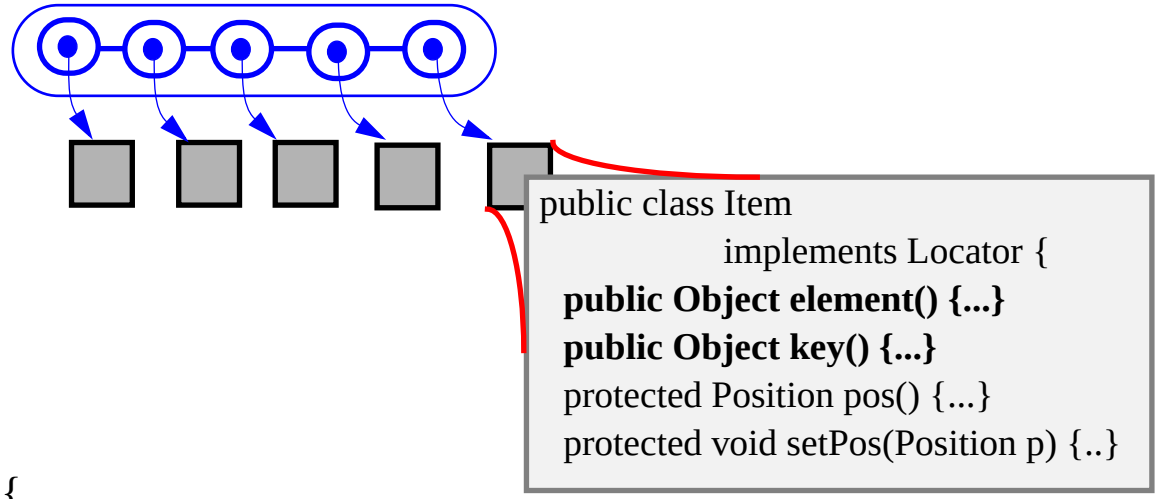
- Item i S kan sammenlignes med cp
- S er usortert / S er sortert

II. Implementasjon av Dictionary – *med Sequence*

I DATA REPRESENTASJON

Sequence, der hver Position

lagrer en Item



II DATA STRUKTUR

```
class DicSequence implements Dictionary {
    private Sequence S;
    private Comparator cp;
    /** @param sq bør være tom sekvens
        @param c Comparator for sammenlikning av Item med hensyn til key-objekter */
    public DicSequence(Sequence sq, Comparator c) {
      S = sq; cp = c; }
```

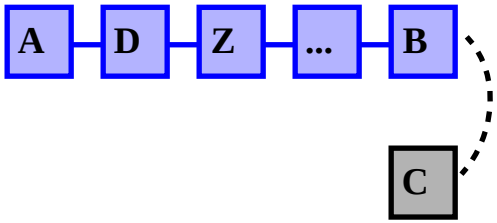
III DATA INVARIANT

- Item i S kan sammenlignes med cp
- S er usortert / S er sortert

cf. PrioritetsKø ....

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

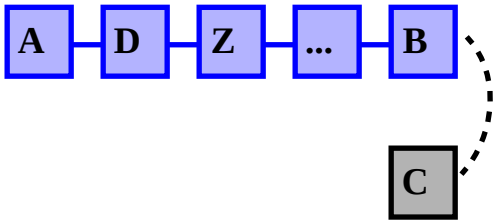
DATA INVARIANT: INGEN – USORTERT




```
void insert (k,o)
   sq.insertLast ( new Item(k,o) );
Object find (Object k):
   Item ko = new Item(k,null);
   Object ret; boolean fant = false;
   Position p = sq.first();
   try{ while ( !fant )  {
      if (cp.isEqualTo(p.element(), ko)
      { ret = ( (Item)p.element() ).element();
         fant = true; }
      else p = sq.after(p);
   } }
   catch (BoundaryExc e) { ret = null;}
   return ret;
```

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) **O**(1)

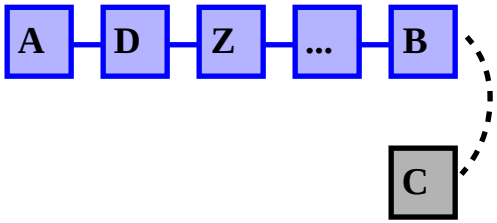
```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): **O(n)**

```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) **O**(1)

```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): **O(n)**

```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

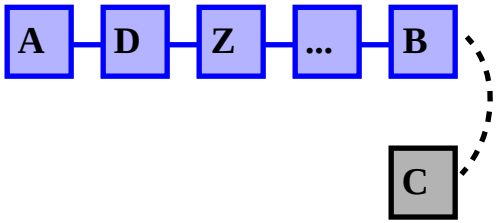
Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **Θ(n)**

Object *closestAfter/Before* (Object k) : find ... **Θ(n)**

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) **O**(1)

```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): **O(n)**

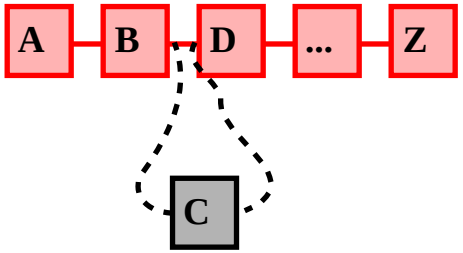
```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **Θ(n)**

Object *closestAfter/Before* (Object k) : find ... **Θ(n)**

DATA INVARIANT: SORTERT STIGENDE

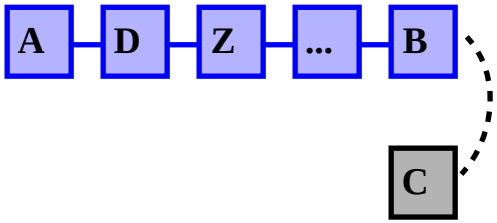


void *insert* (k,o) **O(n)**

```
Item ny= new Item(k,o);
if ( sq.isEmpty() )
   sq.insertFirst(ny)
else if ( cp.isLessOrEqual(ny, sq.first().element()) )
   sq.insertFirst(ny)
else if ( cp.isGreaterOrEqual(ny, sq.last().element()) )
   sq.insertLast(ny)
else Position c = sq.first()
   while (cp.isGreaterThan(ny, c.element() )
      c= sq.after(c)
   sq.insertBefore(c,ny)
```

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) **O**(1)

```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): **O(n)**

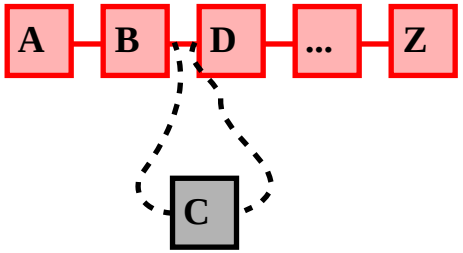
```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **Θ(n)**

Object *closestAfter/Before* (Object k) : find ... **Θ(n)**

DATA INVARIANT: SORTERT STIGENDE



void *insert* (k,o) **O(n)**

```
Item ny= new Item(k,o);
if ( sq.isEmpty() )
   sq.insertFirst(ny)
else if ( cp.isLessOrEqual(ny, sq.first().element()) )
   sq.insertFirst(ny)
else if ( cp.isGreaterOrEqual(ny, sq.last().element()) )
   sq.insertLast(ny)
else Position c = sq.first()
   while (cp.isGreaterThan(ny, c.element() )
      c= sq.after(c)
   sq.insertBefore(c,ny)
```

Object *find* (Object k) : ... **O(n)**

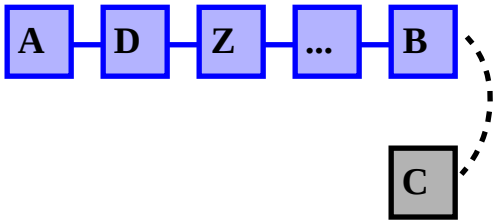
Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **O(n)**

Object *closestAfter/Before* (Object k) : find ... **O(n)**

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) — **O**(1)

```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): — **O(n)**

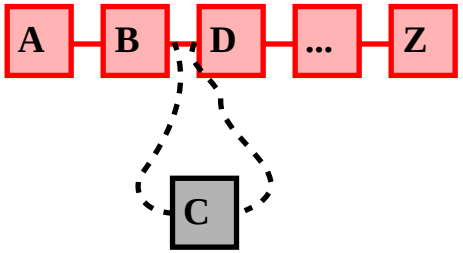
```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

Object *remove* (Object k) : find ... — **O(n)**

ObjectIterator *findAll* (Object k) : find ... — **Θ(n)**

Object *closestAfter/Before* (Object k) : find ... — **Θ(n)**

DATA INVARIANT: SORTERT STIGENDE



void *insert* (k,o) — **O(n)**

```
Item ny= new Item(k,o);
if ( sq.isEmpty() )
   sq.insertFirst(ny)
else if ( cp.isLessOrEqual(ny, sq.first().element()) )
   sq.insertFirst(ny)
else if ( cp.isGreaterOrEqual(ny, sq.last().element()) )
   sq.insertLast(ny)
else Position c = sq.first()
   while (cp.isGreaterThan(ny, c.element() )
      c= sq.after(c)
   sq.insertBefore(c,ny)
```

Object *find* (Object k) : ... — **O(n)**

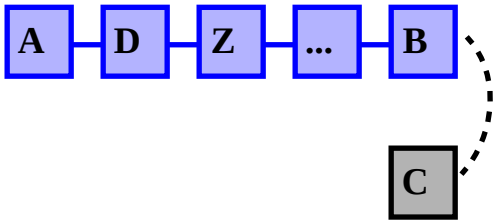
Object *remove* (Object k) : find ... — **O(n)**

ObjectIterator *findAll* (Object k) : find ... — **O(n)**

Object *closestAfter/Before* (Object k) : find ... — **O(n)**

Implementasjon av Dictionary – *med Sequence* *(med LL/DL)*

DATA INVARIANT: INGEN – USORTERT



void *insert* (k,o) **O**(1)

```
sq.insertLast ( new Item(k,o) );
```

Object *find* (Object k): **O(n)**

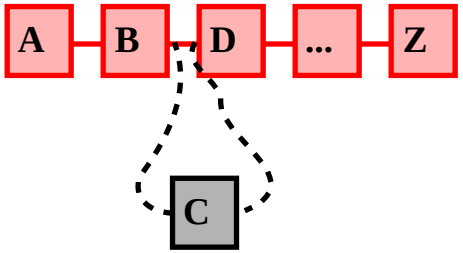
```
Item ko = new Item(k,null);
Object ret; boolean fant = false;
Position p = sq.first();
try{ while ( !fant )  {
   if (cp.isEqualTo(p.element(), ko)
   { ret = ( (Item)p.element() ).element();
      fant = true; }
   else p = sq.after(p);
} }
catch (BoundaryExc e) { ret = null;}
return ret;
```

Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **Θ(n)**

Object *closestAfter/Before* (Object k) : find ... **Θ(n)**

DATA INVARIANT: SORTERT STIGENDE



void *insert* (k,o) **O(n)**

```
Item ny= new Item(k,o);
if ( sq.isEmpty() )
   sq.insertFirst(ny)
else if ( cp.isLessOrEqual(ny, sq.first().element()) )
   sq.insertFirst(ny)
else if ( cp.isGreaterOrEqual(ny, sq.last().element()) )
   sq.insertLast(ny)
else Position c = sq.first()
   while (cp.isGreaterThan(ny, c.element() )
      c= sq.after(c)
   sq.insertBefore(c,ny)
```

Object *find* (Object k) : ... **O(n)**

Object *remove* (Object k) : find ... **O(n)**

ObjectIterator *findAll* (Object k) : find ... **O(n)**

Object *closestAfter/Before* (Object k) : find ... **O(n)**

PriorityQueue

Object *removeMin* () : find ... **O(n)**

Object *removeMin* () : **O(1)**

Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

< **cp.isLessThan**

[0]	11
[1]	19
[2]	24
[3]	32
[4]	32
[5]	48
[6]	50
[7]	55
[8]	72
[9]	99

Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

< **cp.isLessThan**

[0]	11
[1]	19
[2]	24
[3]	32
[4]	32
[5]	48
[6]	50
[7]	55
[8]	72
[9]	99

Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

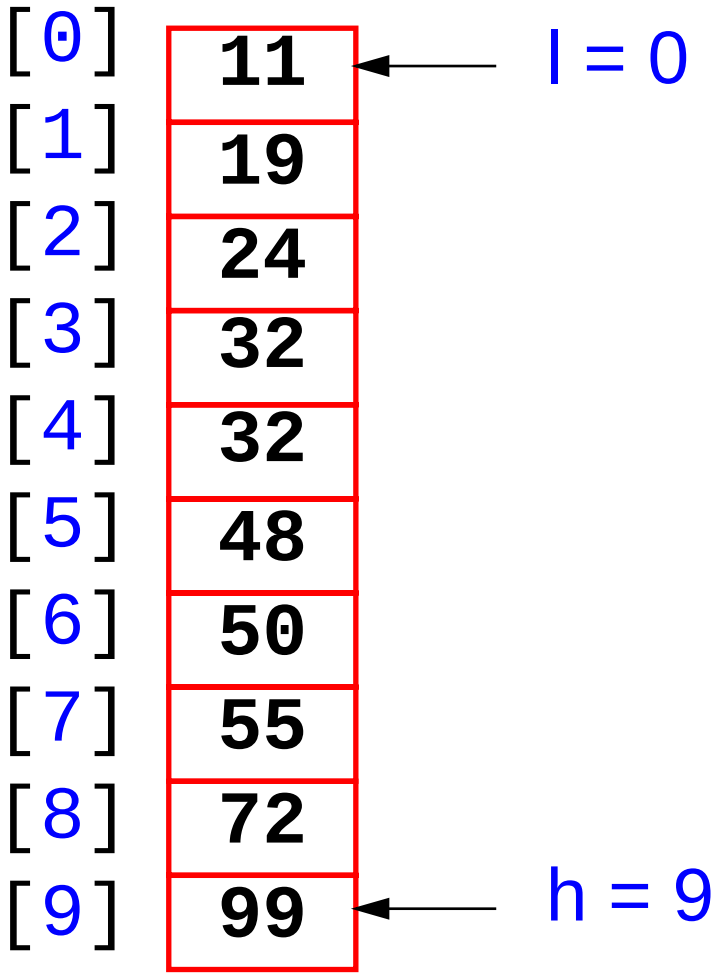
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

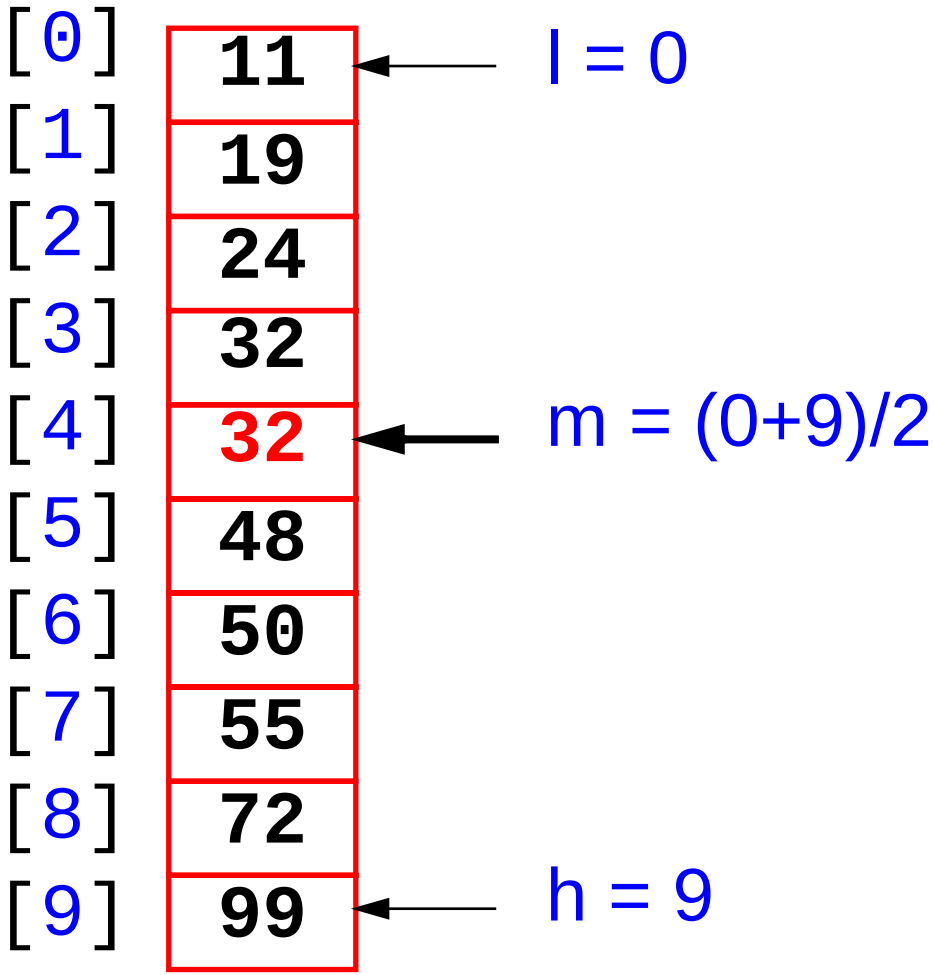
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

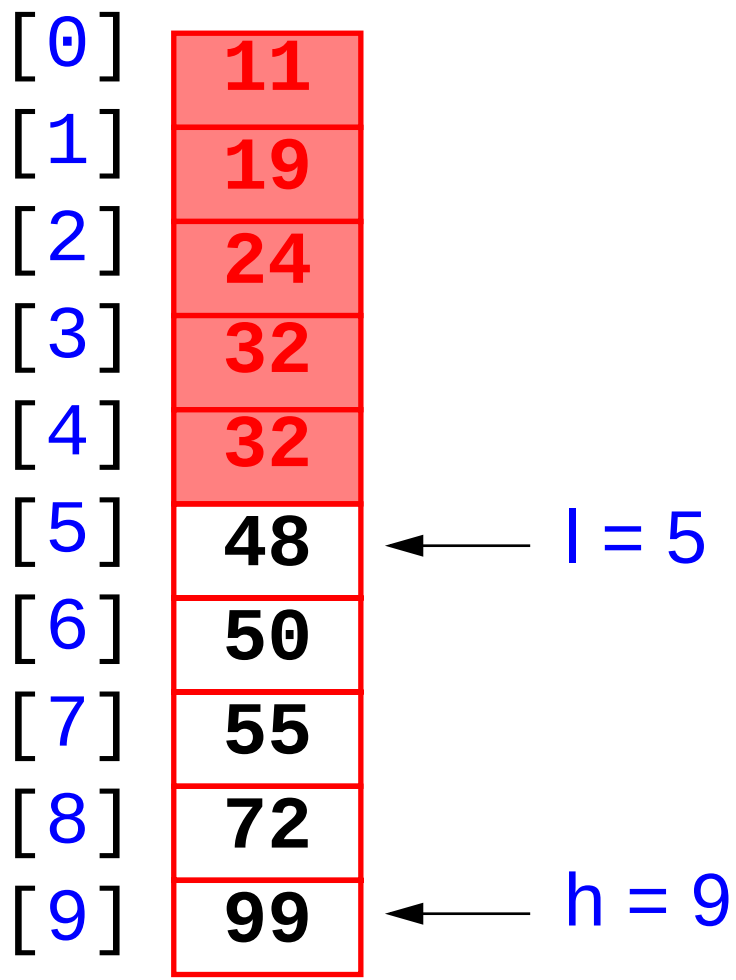
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

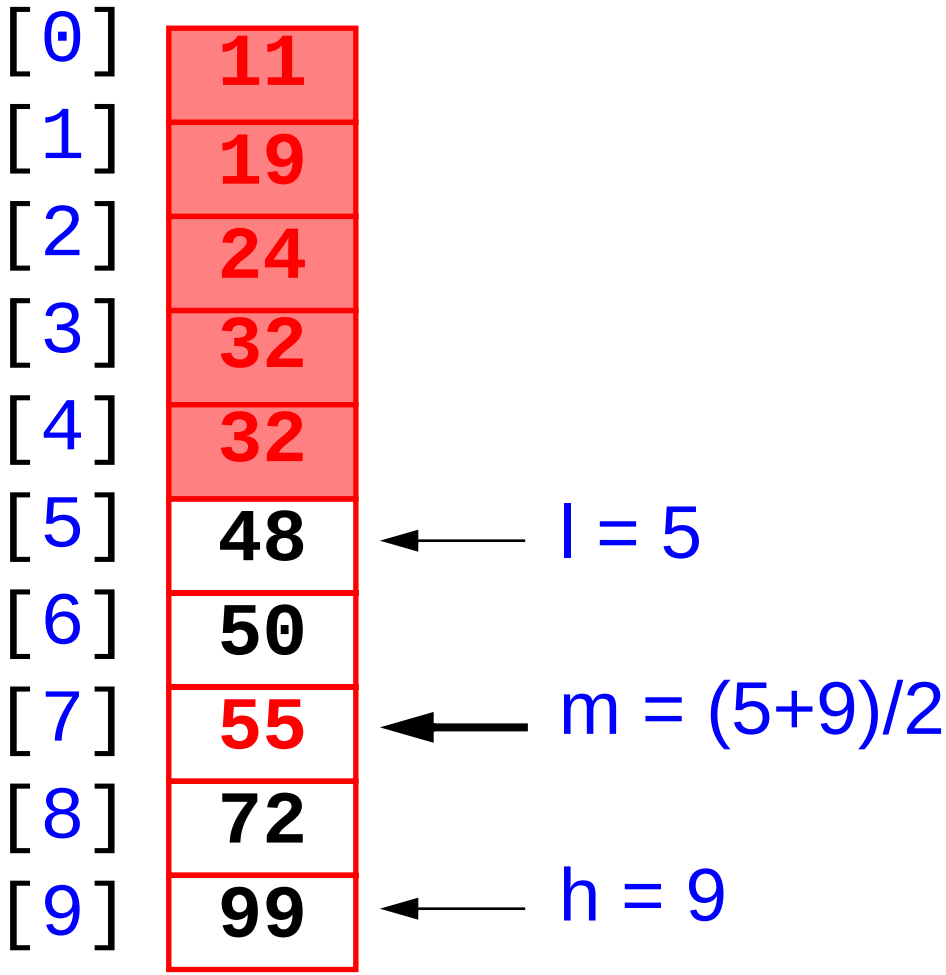
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

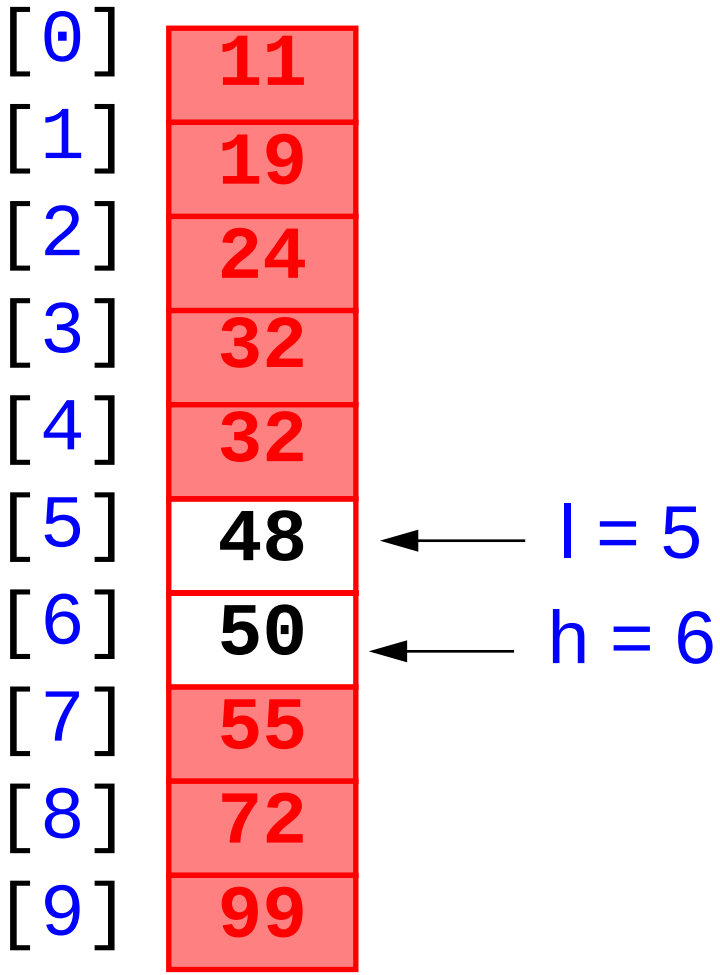
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

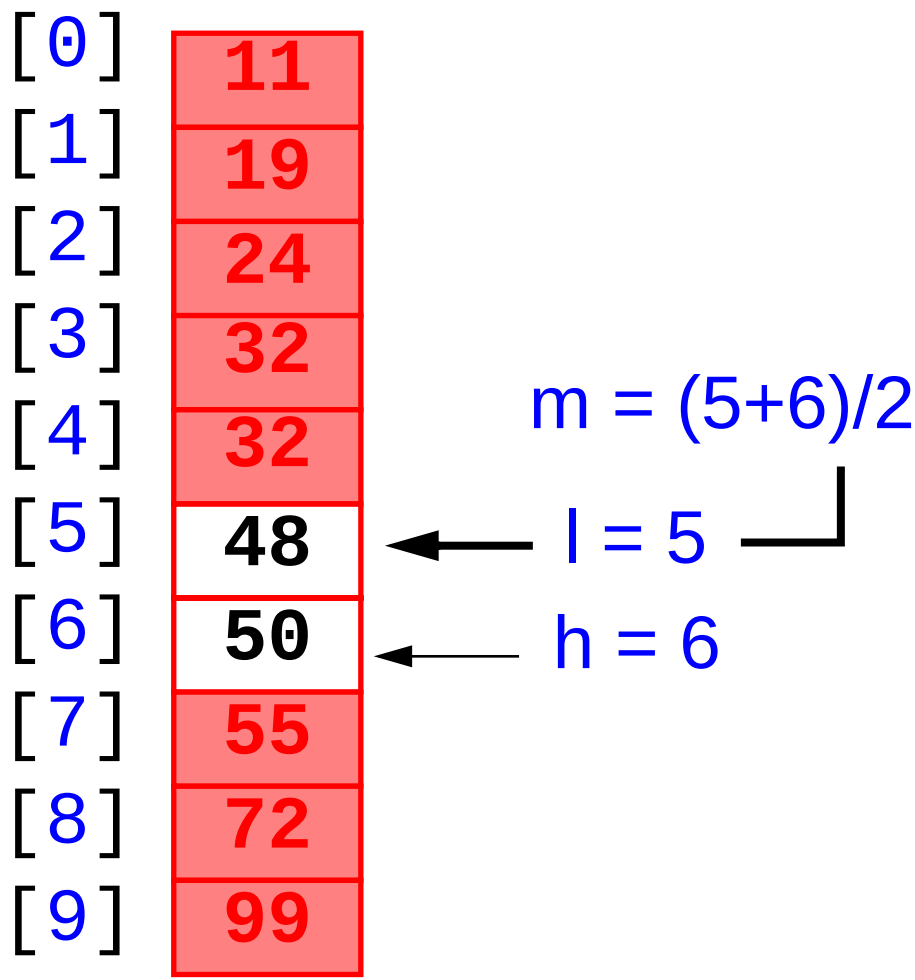
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

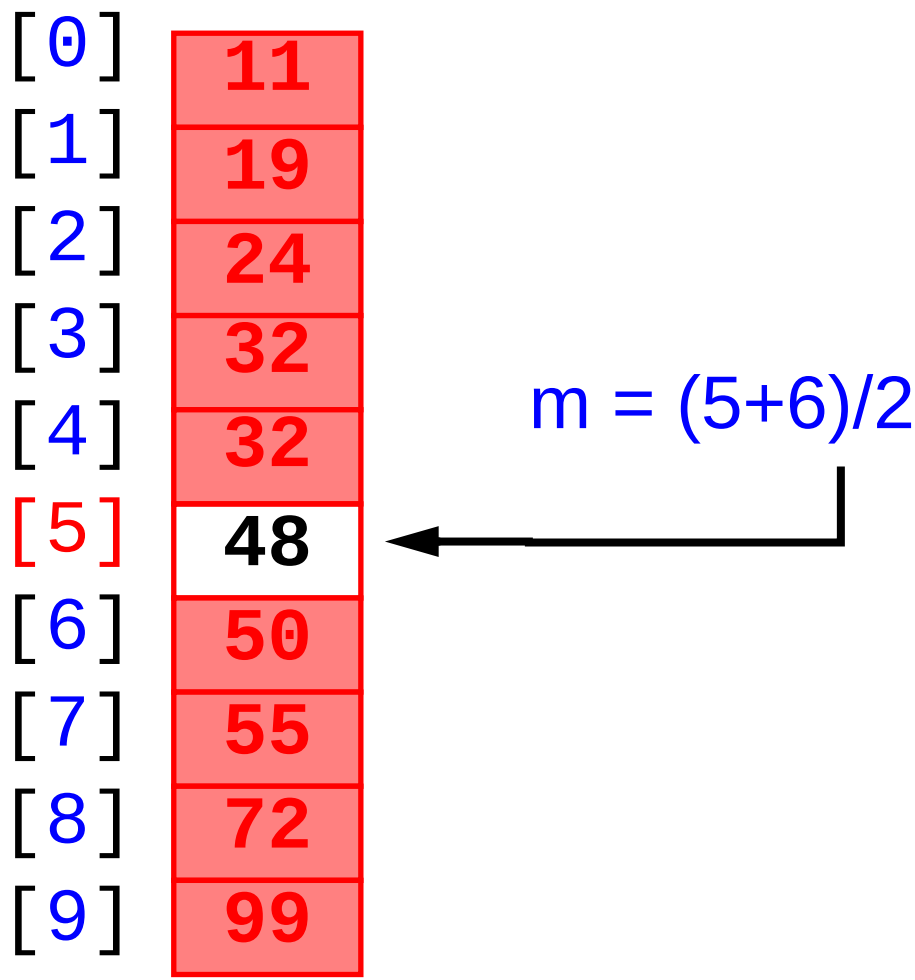
  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary – *med Sequence* *(med Array)*

III. DATA INVARIANT :

Sortert Array A[l...h] :

- **hvis** i < j **så** **cp.isLessOrEqual**(A[i], A[j])
- **hvis** **cp.isLessThan**(A[i], A[j]) **så** i < j

  hvis **cp.isGreaterThan**(A[i], A[j]) **så** i > j

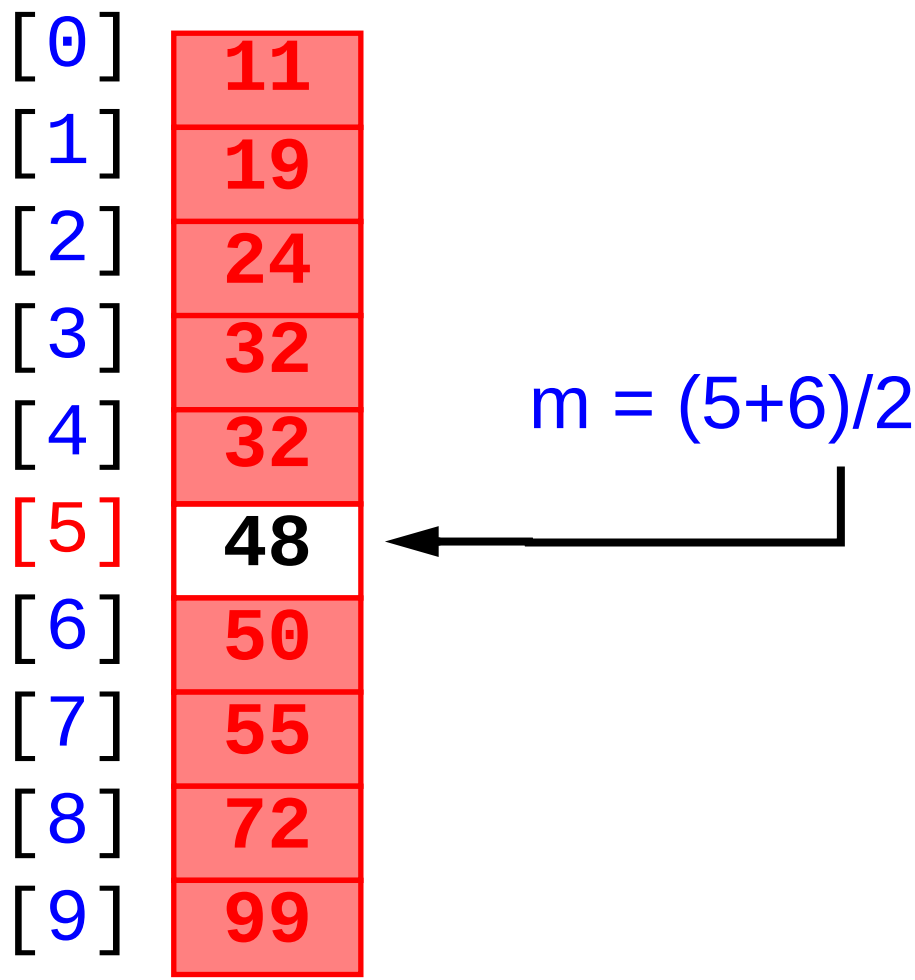
BinærSøk

```
find(k) : find(k, l, h)
  if ( l > h ) return -1
  else
      m = (l+h) / 2
      if ( cp.isEqualTo(A[m],k) )
        return A[m].element();
      else if ( cp.isLessThan(A[m],k) )
        return finn(k,m+1,h)
      else //  cp.isGreaterThan(A[m],k)
        return finn(k,l,m-1)
```

find(k) = **BinærSøk(k)** = **O(log n)**

find (**48**)

< **cp.isLessThan**



Implementasjon av Dictionary (så langt)

	usortert **Sequence**		sortert **Sequence**	
operasjon	**Array**	**DL**	**Array**	**DL**
Container				
size	1	1	1	1
isEmpty	1	1	1	1
elements	n	n	n	n
Dictionary				
find	n	n	log n	n
findAll	n	n	log n (+ s)	n
insert	1	1	n	n
remove	n	n	n	n
closestAfter/Before	n	n	n	n

III. Implementasjon av Dictionary – *med HashTabell*

key : elementer	→	**nøkkel [heltall]**	**injektiv**	**TO**
Personer	→	personnummer	+	+
Personer	→	fødselsår	–	+
variable i et program	→	type (int, boolean, Tree...)	–	–

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }

III. Implementasjon av Dictionary – *med HashTabell*

key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

III. Implementasjon av Dictionary – *med HashTabell*

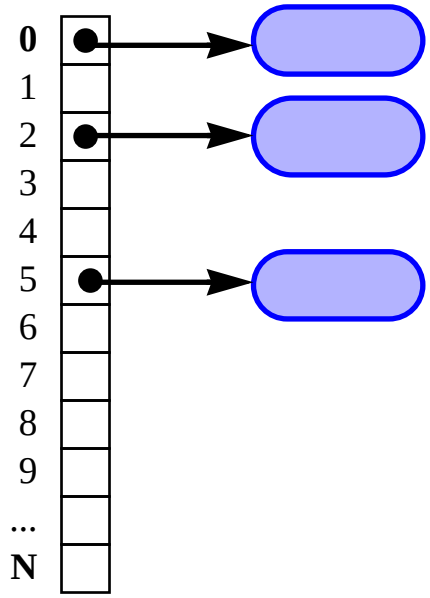
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

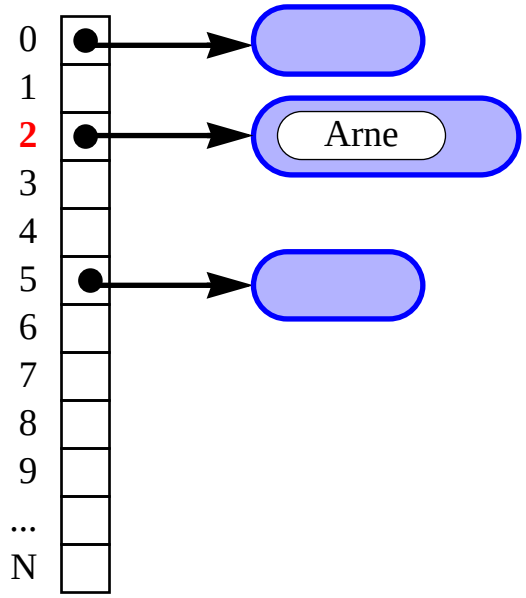
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972)

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

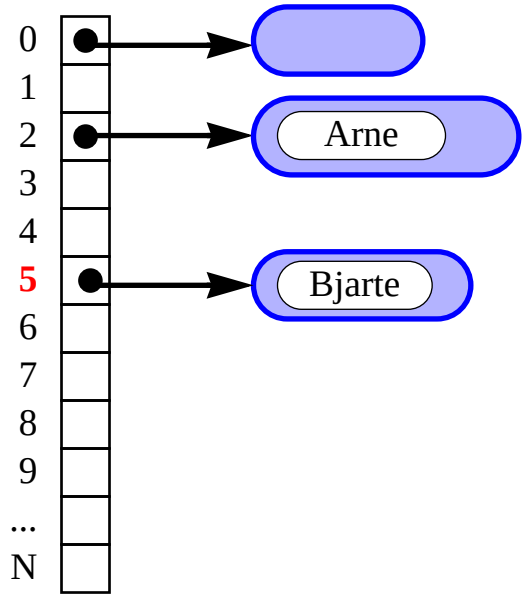
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), **Bjarte(1975)**

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

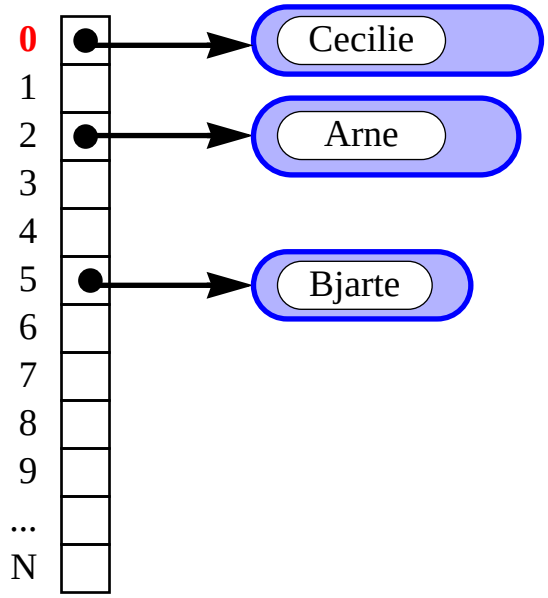
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), **Cecilie(1970)**

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

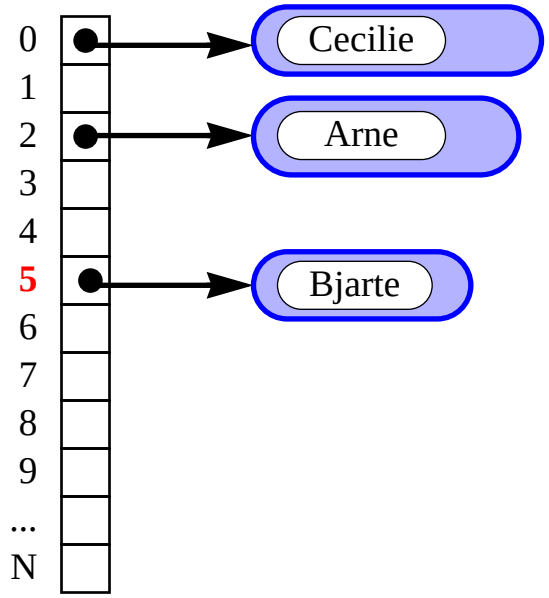
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), **Dagmar(1975)**

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

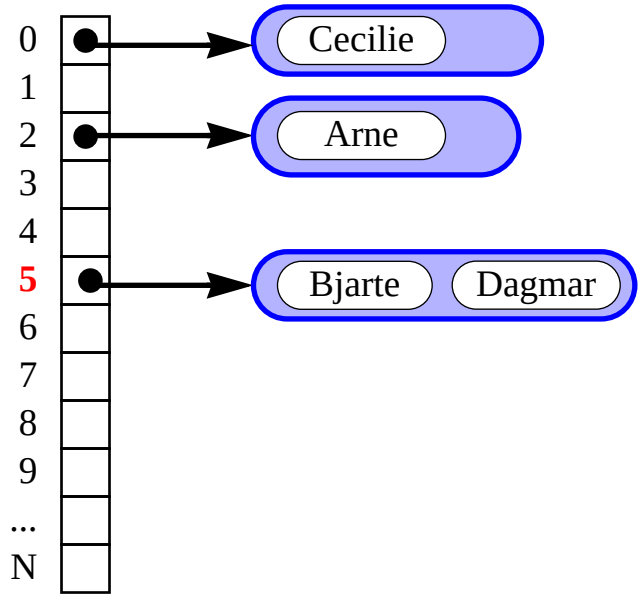
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), **Dagmar(1975)**

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

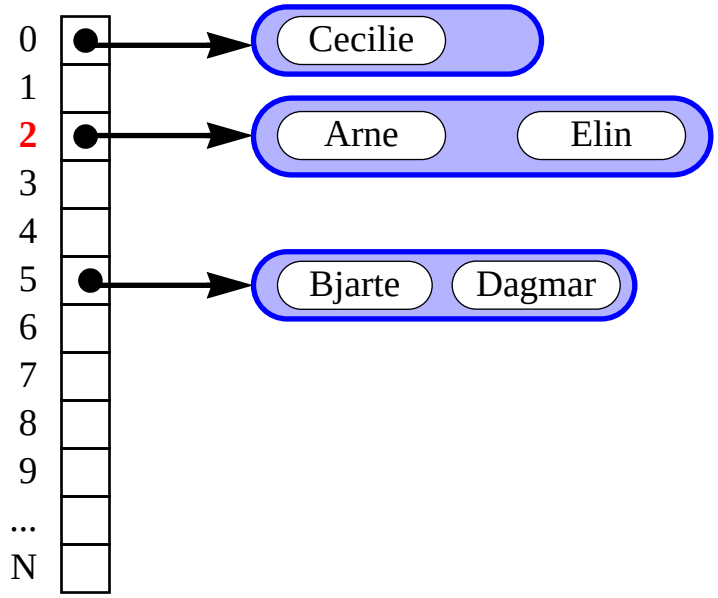
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), **Elin(1972)**

int key(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

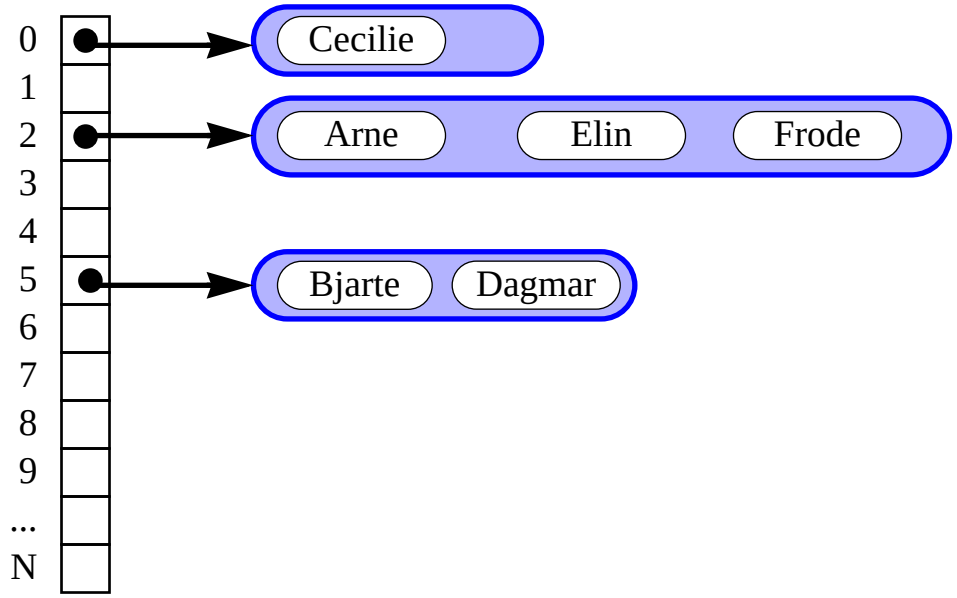
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), **Frode(1972)**

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



III. Implementasjon av Dictionary – *med HashTabell*

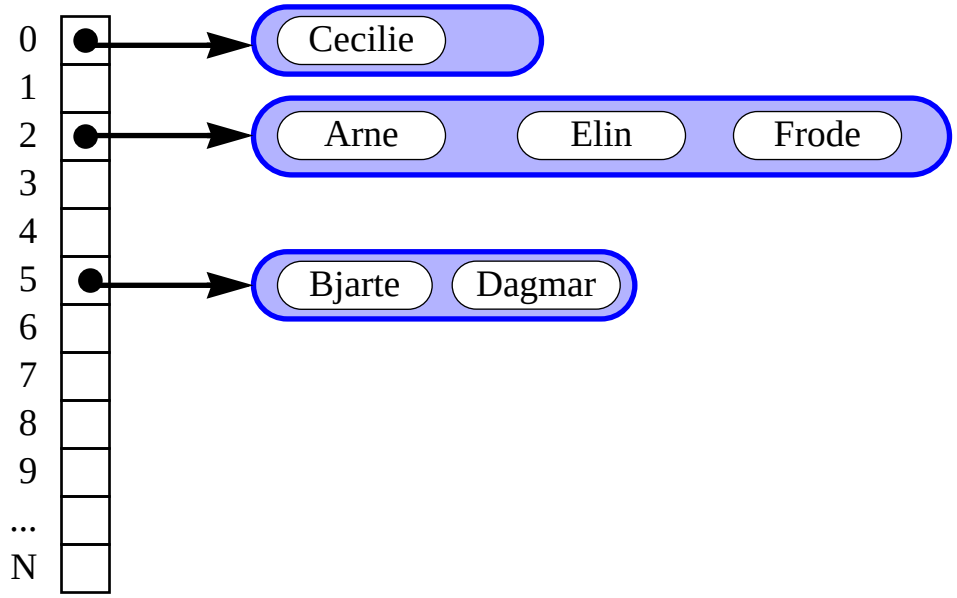
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



insert(int k, Object o) {
H[**hash**(k)].insertFirst(o) }

III. Implementasjon av Dictionary – *med HashTabell*

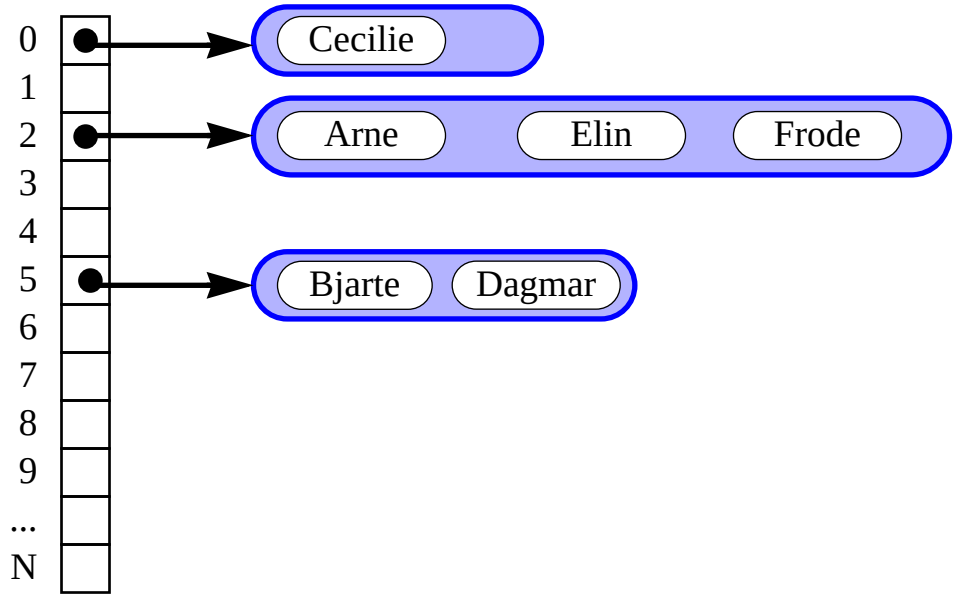
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



insert(int k, Object o) { $O(1)$
H[**hash**(k)].insertFirst(o) }

III. Implementasjon av Dictionary – *med HashTabell*

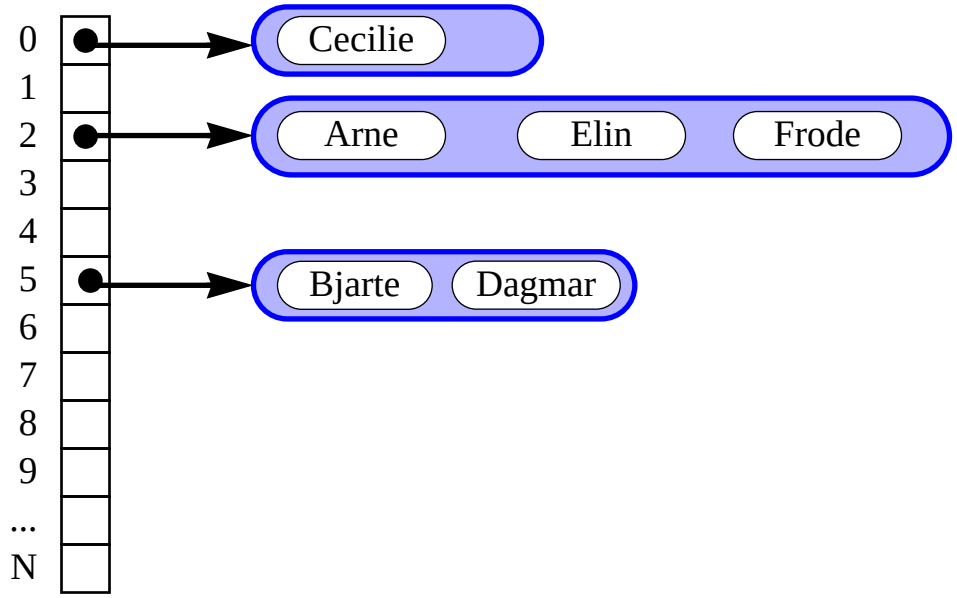
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



insert(int k, Object o) { $O(1)$
H[**hash**(k)].insertFirst(o) }

find(int k) { $O(1)$
Sequence S = H[**hash**(k)]
return S.first().element() }

III. Implementasjon av Dictionary – *med HashTabell*

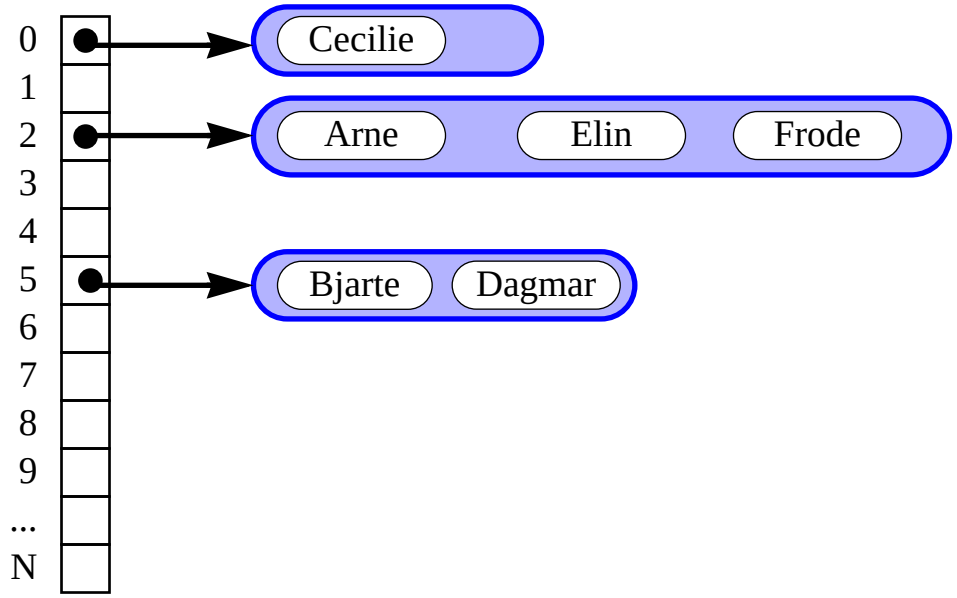
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



insert(int k, Object o) { $O(1)$
H[**hash**(k)].insertFirst(o) }

find(int k) { $O(1)$
Sequence S = H[**hash**(k)]
return S.first().element() }

remove(int k) { $O(1)$
Sequence S = H[**hash**(k)]
S.remove(S.first())}

III. Implementasjon av Dictionary – *med HashTabell*

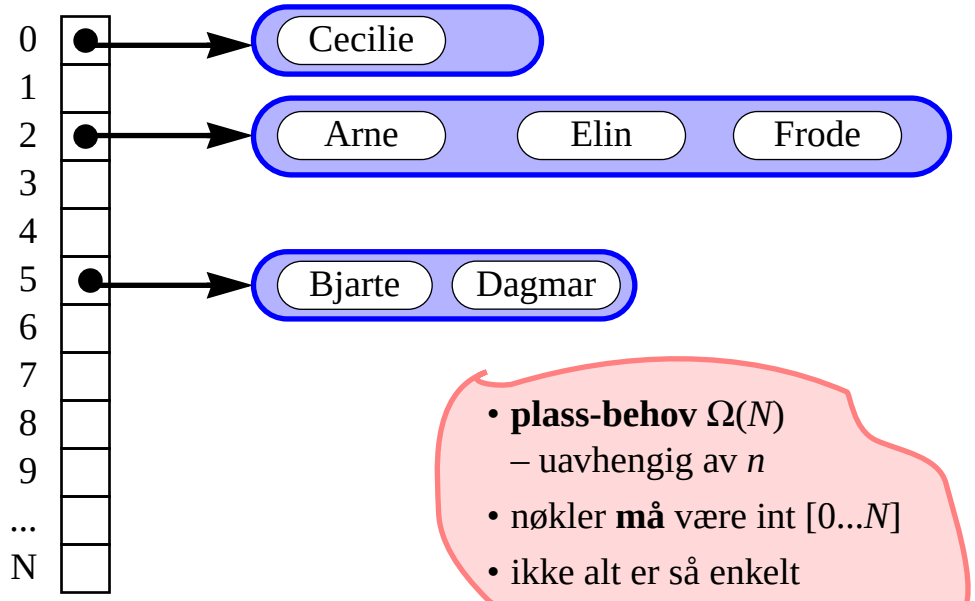
key : elementer	→	**nøkkel [heltall]**	**hash**	**heltall [0 ... N]**
Personer	→	personnummer	→	??
Personer	→	fødselsår	→	??
variable i et program	→	type (int, boolean, Tree...)	→	??

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1975), Elin(1972), Frode(1972)

int **key**(Person p) { return p.fødselsår }
int **hash**(int k) { return k – 1970 }

key :	**elementer**	→ **nøkkel**
hash :		int **nøkkel** → int **hashkode**

```
Sequence[] H = new Sequence[N]
```



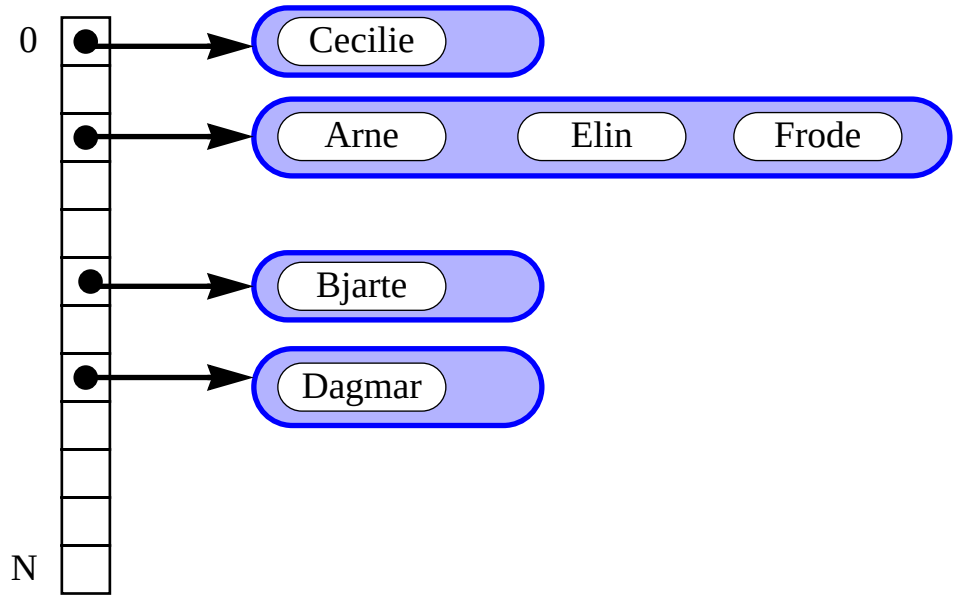
- **plass-behov** $\Omega(N)$
  – uavhengig av n
- nøkler **må** være int $[0...N]$
- ikke alt er så enkelt

insert(int k, Object o) { $O(1)$
  H[**hash**(k)].insertFirst(o) }

find(int k) { $O(1)$
  Sequence S = H[**hash**(k)]
  return S.first().element() }

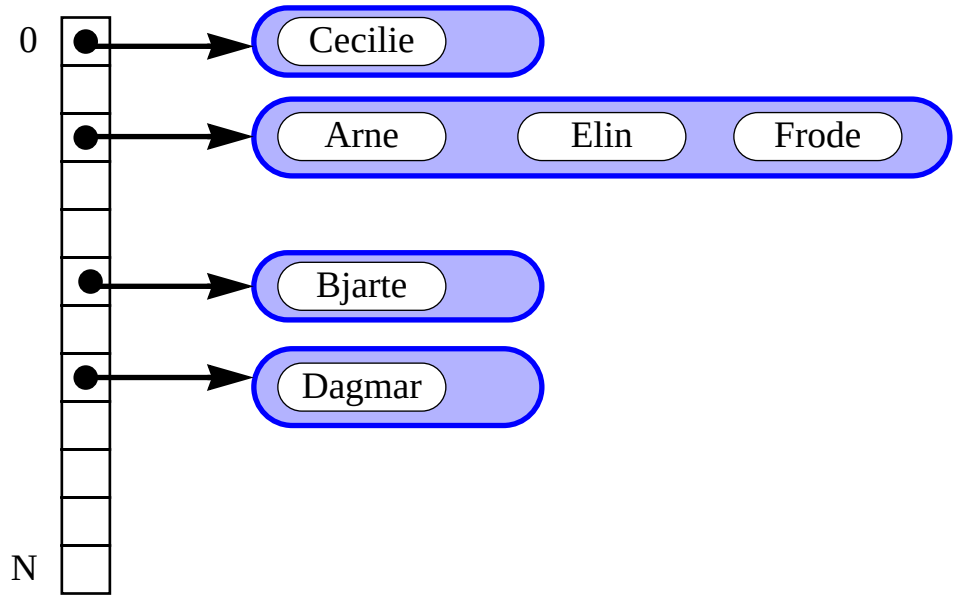
remove(int k) { $O(1)$
  Sequence S = H[**hash**(k)]
  S.remove(S.first())}

```
Sequence[] H = new Sequence[N]
```




```
int hash(int k) { return k – 1970 }

insert(int k, Object o) {
    H[hash(k)].insertFirst(o) }          O(1)

find(int k) {
    Sequence S = H[ hash(k) ]
    return S.first().elem() }            O(1)

remove(int k) {
    Sequence S = H[ hash(k) ]
    S.remove(S.first())}                 O(1)
```

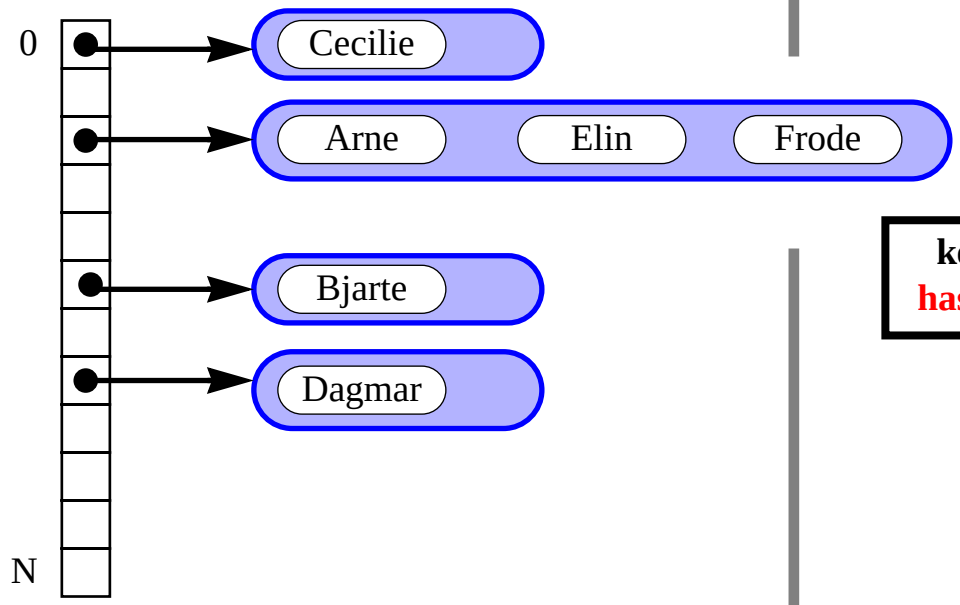
```
Sequence[] H = new Sequence[N]
```




```
int hash(int k) { return k – 1970 }

insert(int k, Object o) {
    H[hash(k)].insertFirst(o) }          O(1)

find(int k) {
    Sequence S = H[ hash(k) ]
    return S.first().elem() }            O(1)

remove(int k) {
    Sequence S = H[ hash(k) ]
    S.remove(S.first())}                 O(1)
```

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

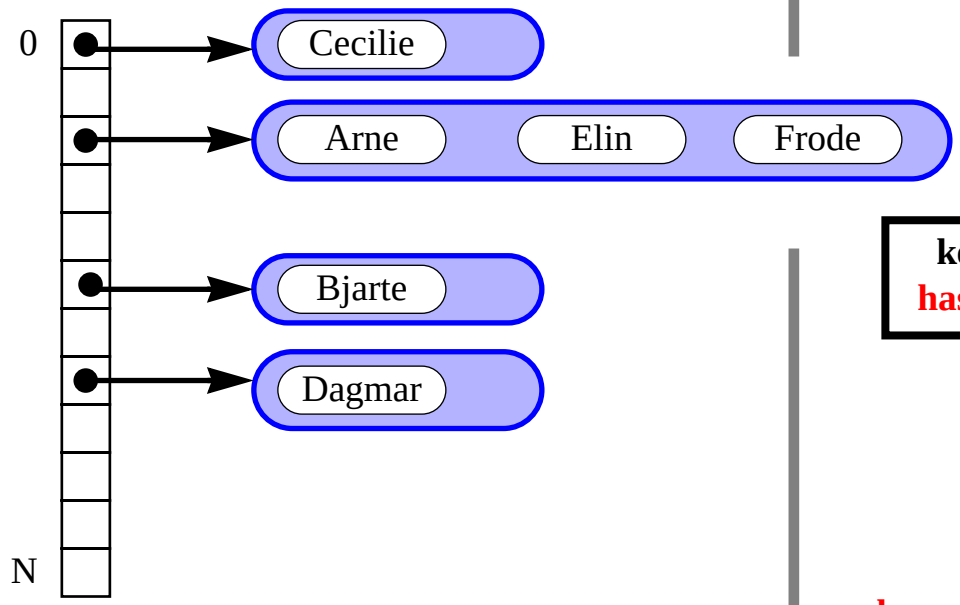
Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : **personer** → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

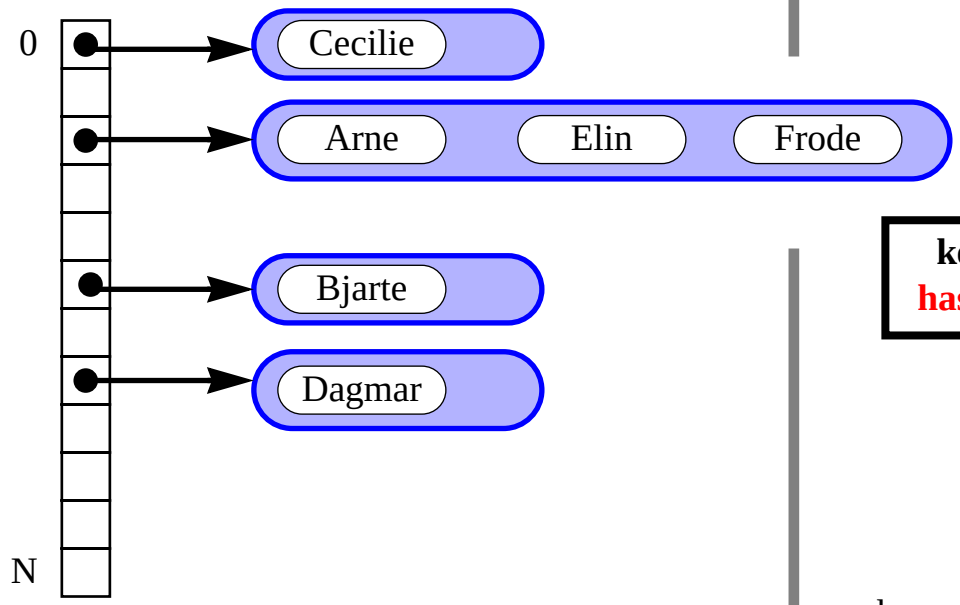
Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : **personer** → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

key er injektiv men hash er det ikke

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

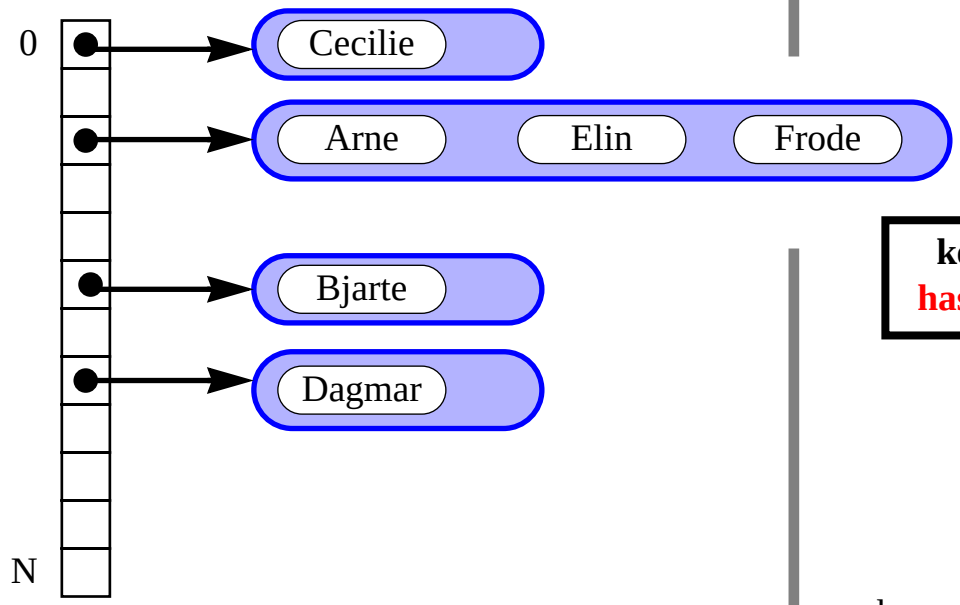
key : **personer** → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

key er injektiv men hash er det ikke

insert(int **pn**, Object o) {
H[**hash**(**pn**)].insertFirst(o) } $O(1)$

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : **personer** → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

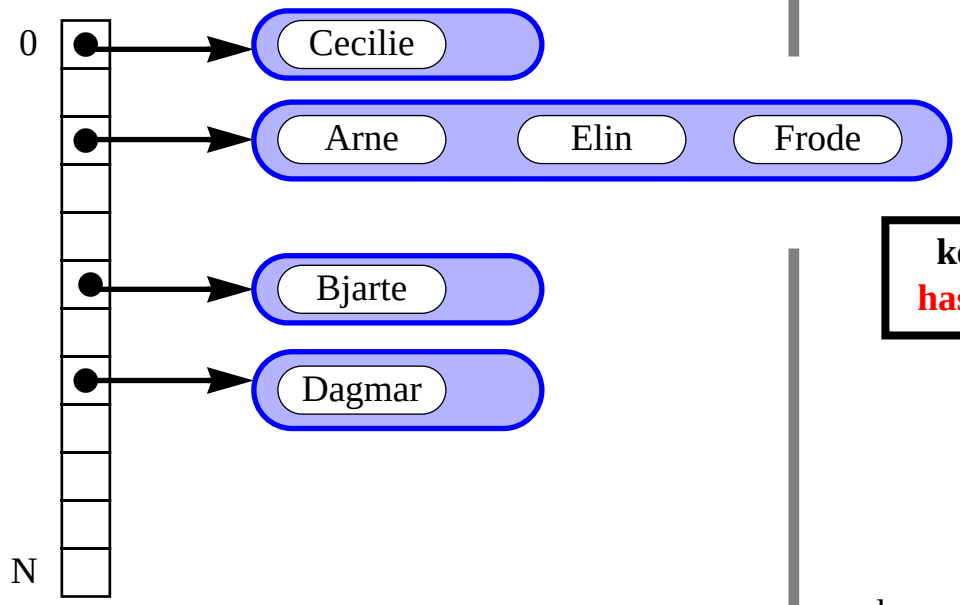
Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

key er injektiv men hash er det ikke

insert(int **pn**, Object o) {
H[**hash**(**pn**)].insertFirst(o) } $O(1)$

find(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S }

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : **personer** → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

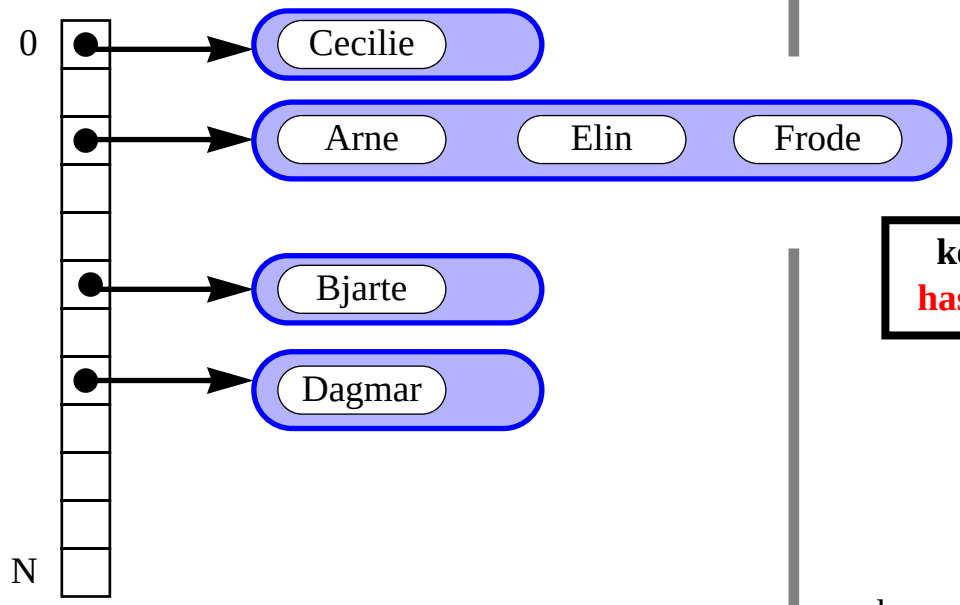
key er injektiv men hash er det ikke

insert(int **pn**, Object o) {
H[**hash**(**pn**)].insertFirst(o) } $O(1)$

find(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S } $O(|S|)$

remove(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S og fjern } $O(|S|)$

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : personer → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

key er injektiv men hash er det ikke

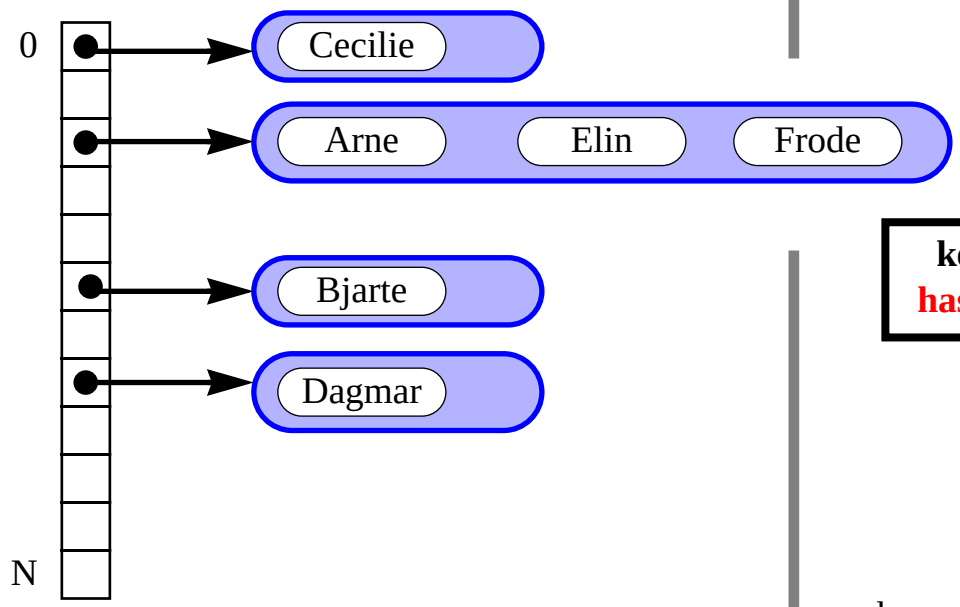
insert(int **pn**, Object o) {
H[**hash**(**pn**)].insertFirst(o) } $O(1)$

find(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S } $O(|S|)$

remove(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S og fjern } $O(|S|)$

GENERELT: hash er ikke injektiv
forventet/gjennomsnittlig **|S| = load factor = n/N**

```
Sequence[] H = new Sequence[N]
```



int **hash**(int k) { return k – 1970 }

insert(int **k**, Object o) {
H[**hash**(**k**)].insertFirst(o) } $O(1)$

find(int **k**) {
Sequence S = H[**hash**(**k**)]
return S.first().elem() } $O(1)$

remove(int **k**) {
Sequence S = H[**hash**(**k**)]
S.remove(S.first())} $O(1)$

injektiv hash:
hash(k1) = hash(k2) hviss k1 = k2

Et typisk eksempel ...

Nøkkel er personnummer og vi skal samle alle ansatte.
Vi kan ikke indeksere tabellen med persnr
– det er jo bare 250 ansatte!!

key : personer → **persnr**
hash : int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	212131 92345	2345

key er injektiv men hash er det ikke

insert(int **pn**, Object o) {
H[**hash**(**pn**)].insertFirst(o) } $O(1)$

find(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S } $O(|S|)$

remove(int **pn**) {
Sequence S = H[**hash**(**pn**)]
finn pn i S og fjern } $O(|S|)$

GENERELT: hash er ikke injektiv
forventet/gjennomsnittlig **|S| = load factor = n/N < 1**

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

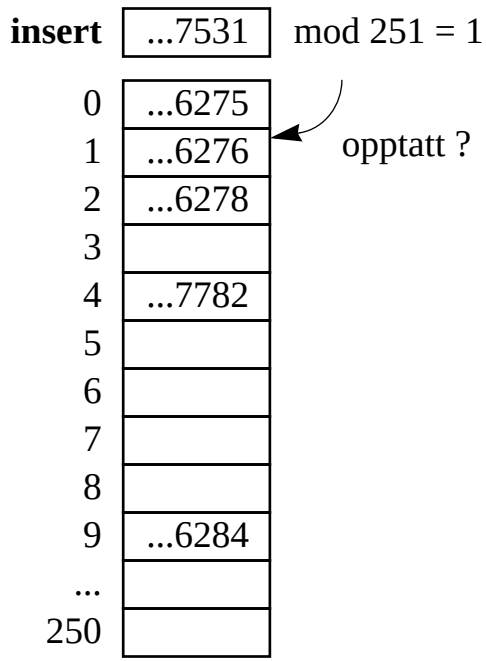
hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```

insert	...7531	mod 251 = 1
0	...6275	
1	...6276	
2	...6278	
3		
4	...7782	
5		
6		
7		
8		
9	...6284	
...		
250		

Linear Probing



Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

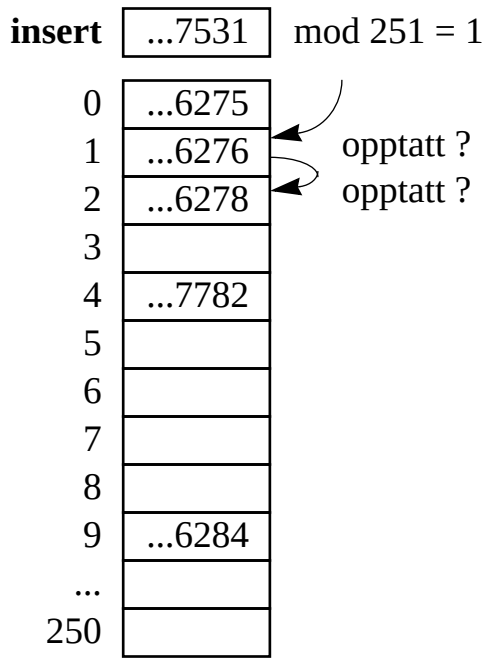
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```

Linear Probing



Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

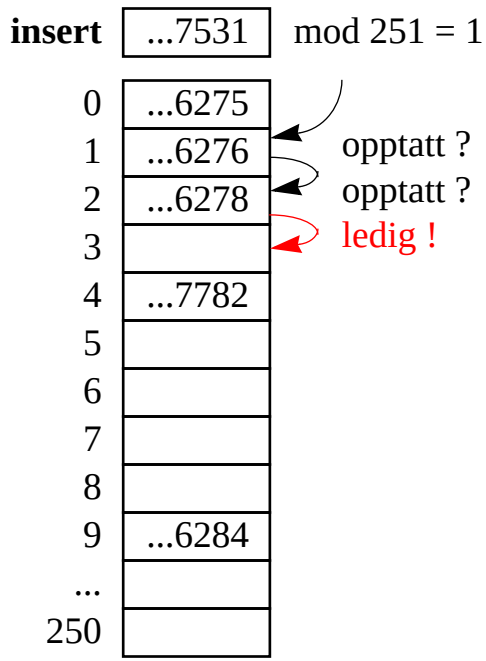
ansatt[] H[250]

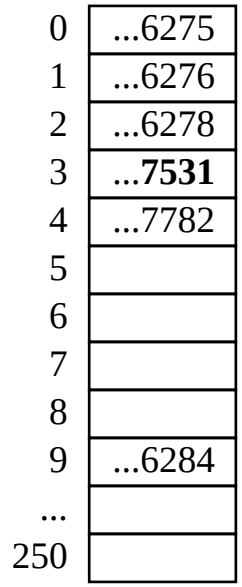
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

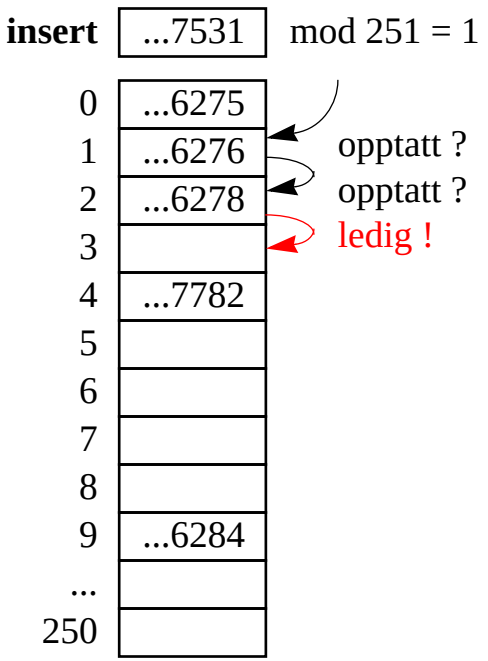
ansatt[] H[250]

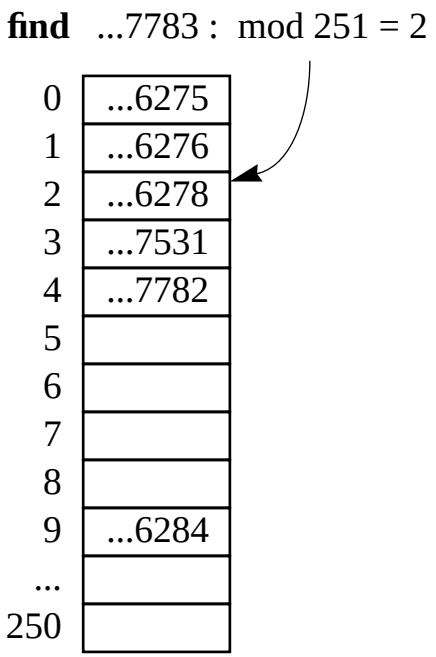
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

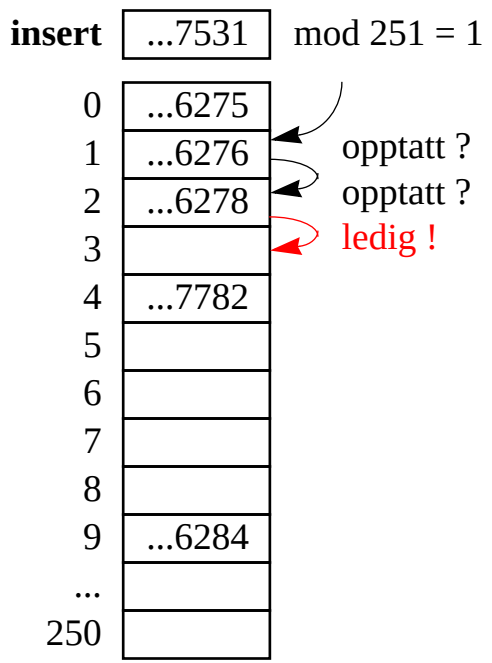
ansatt[] H[250]

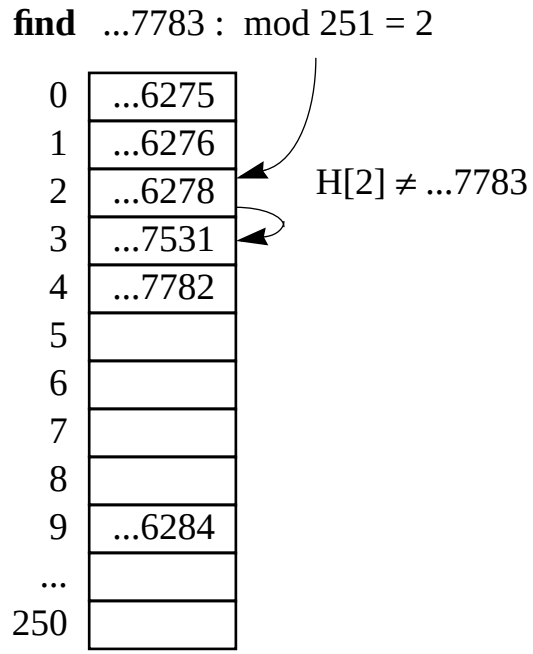
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

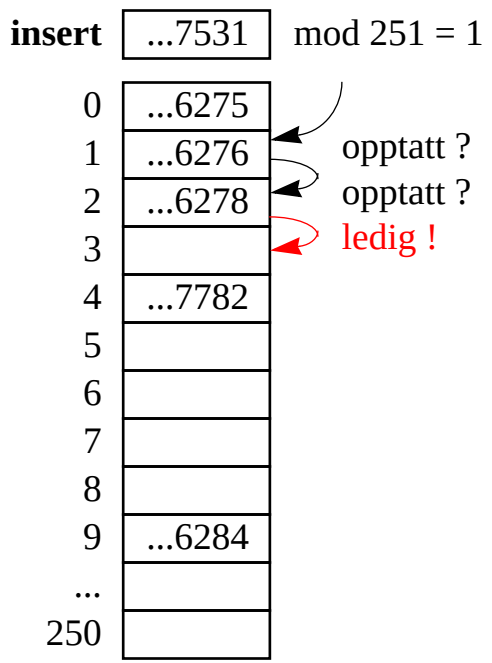
ansatt[] H[250]

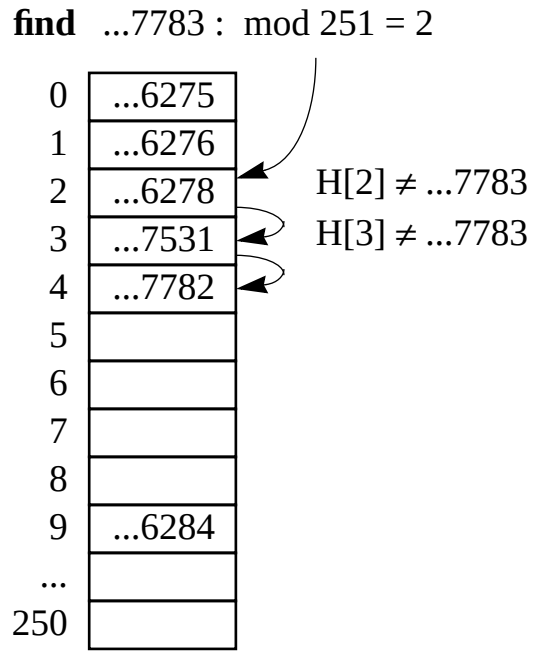
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H**[**251**]

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

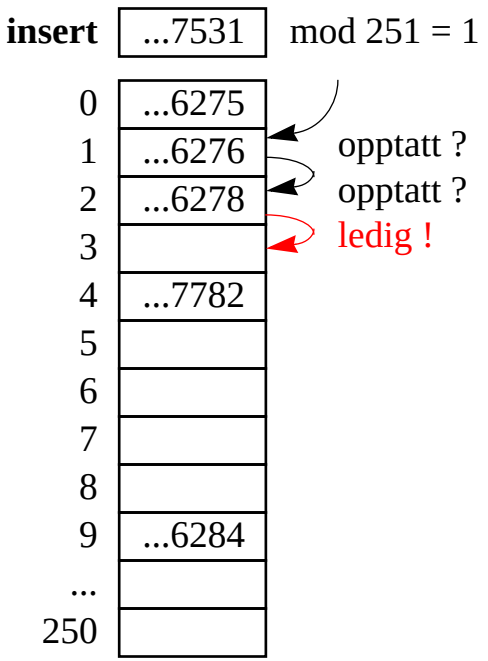
ansatt[] H[250]

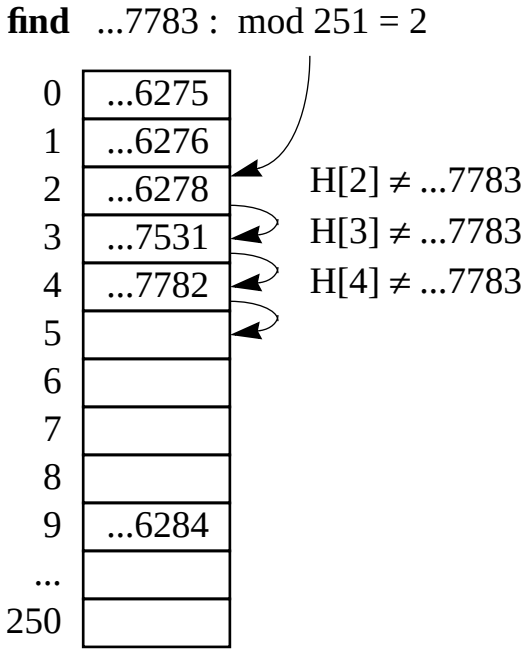
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

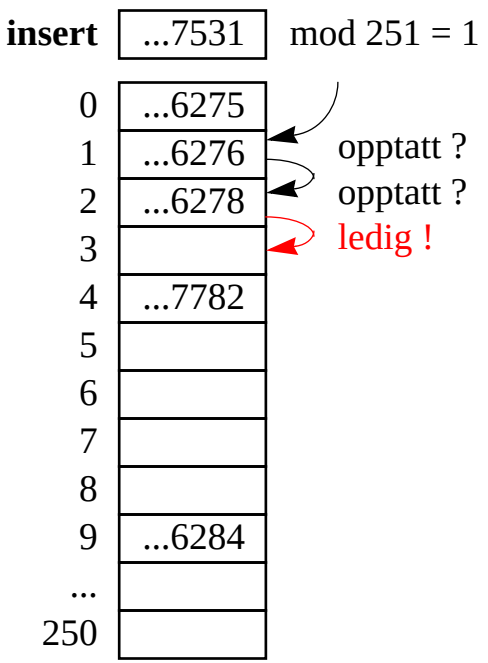
ansatt[] H[250]

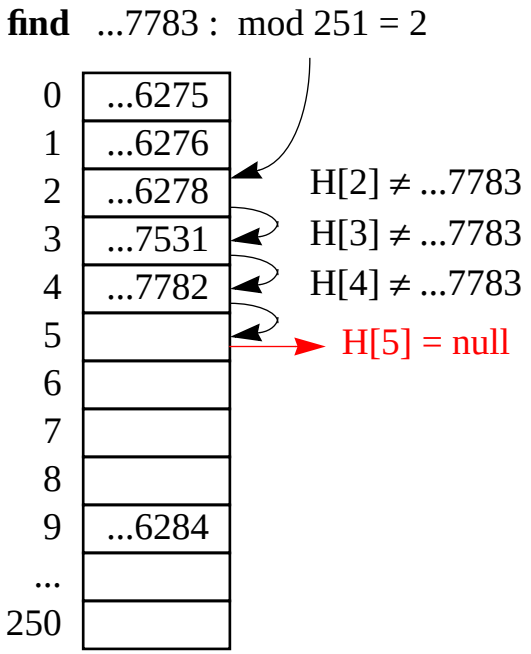
hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

```
int hash(int k) {
  s = siste 4 siffrene fra k
  return (s % 251) }
```





Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = **1**

0	...6275
1	..**6276**
2	...6278
3	..**7531**
4	..**7782**
5	...7535
6	...**8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

H[1] ≠ ...7531

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275	
1	...6276	opptatt ?
2	...6278	opptatt ?
3		ledig !
4	...7782	
5		
6		
7		
8		
9	...6284	
...		
250		

find ...7783 : mod 251 = 2

0	...6275	
1	...6276	
2	...6278	H[2] ≠ ...7783
3	...7531	H[3] ≠ ...7783
4	...7782	H[4] ≠ ...7783
5		H[5] = null
6		
7		
8		
9	...6284	
...		
250		

remove ...7531 : mod 251 = 1

0	...6275	
1	**..6276**	H[1] ≠ ...7531
2	...6278	H[2] ≠ ...7531
3	**..7531**	
4	**..7782**	
5	...7535	
6	**...8033**	
7		
8		
9	...6284	
...		
250		

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275	
1	...6276	opptatt ?
2	...6278	opptatt ?
3		ledig !
4	...7782	
5		
6		
7		
8		
9	...6284	
...		
250		

find ...7783 : mod 251 = 2

0	...6275	
1	...6276	
2	...6278	H[2] ≠ ...7783
3	...7531	H[3] ≠ ...7783
4	...7782	H[4] ≠ ...7783
5		H[5] = null
6		
7		
8		
9	...6284	
...		
250		

remove ...7531 : mod 251 = 1

0	...6275	
1	**..6276**	H[1] ≠ ...7531
2	...6278	H[2] ≠ ...7531
3	**..7531**	H[3] = ...7531
4	**..7782**	
5	...7535	
6	**...8033**	
7		
8		
9	...6284	
...		
250		

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

H[1] ≠ ...7531
H[2] ≠ ...7531
H[3] = ...7531

0	...6275
1	**...6276**
2	...6278
3	
4	**...7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

H[1] ≠ ...7531
H[2] ≠ ...7531
H[3] = ...7531

0	...6275
1	**...6276**
2	...6278
3	
4	**...7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275
1	...6276
2	...6278
3	
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

opptatt ?
opptatt ?
ledig !

find ...7783 : mod 251 = 2

0	...6275
1	...6276
2	...6278
3	...7531
4	...7782
5	
6	
7	
8	
9	...6284
...	
250	

H[2] ≠ ...7783
H[3] ≠ ...7783
H[4] ≠ ...7783
H[5] = null

remove ...7531 : mod 251 = 1

0	...6275
1	**..6276**
2	...6278
3	**..7531**
4	**..7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

H[1] ≠ ...7531
H[2] ≠ ...7531
H[3] = ...7531

0	...6275
1	**...6276**
2	...6278
3	
4	**...7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

insert ...7531 mod 251 = 1

0	...6275	
1	...6276	opptatt ?
2	...6278	opptatt ?
3		ledig !
4	...7782	
5		
6		
7		
8		
9	...6284	
...		
250		

find ...7783 : mod 251 = 2

0	...6275	
1	...6276	
2	...6278	H[2] ≠ ...7783
3	...7531	H[3] ≠ ...7783
4	...7782	H[4] ≠ ...7783
5		H[5] = null
6		
7		
8		
9	...6284	
...		
250		

remove ...7531 : mod 251 = 1

0	...6275	
1	**..6276**	H[1] ≠ ...7531
2	...6278	H[2] ≠ ...7531
3	**..7531**	H[3] = ...7531
4	**..7782**	
5	...7535	
6	**...8033**	
7		
8		
9	...6284	
...		
250		

0	...6275
1	**...6276**
2	...6278
3	**søk**
4	**...7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Linear Probing

Ansatte identifseres med
nøkkel = personnr.: 11-siffer heltall

ansatt[] H[98765432101] ....

men det er kun 250 ansatte.

ansatt[] H[250]

hash:12345676275
mod 250 ...
mod 251 ...
siste 4 siffre mod 251

hash(12345676275)
= 6275 mod 251 = 0

ansatt[] **H[251]**

int **hash**(int **k**) {
s = siste 4 siffrene fra k
return (**s % 251**) }

H[(i +1) mod N] , H[(i +2) mod N], ...

H[(i + j^2) mod N], j = 0,1,2...

H[(i + h(j,k)) mod N]

insert ...7531 mod 251 = 1

0	...6275	
1	...6276	opptatt ?
2	...6278	opptatt ?
3		ledig !
4	...7782	
5		
6		
7		
8		
9	...6284	
...		
250		

find ...7783 : mod 251 = 2

0	...6275	
1	...6276	
2	...6278	H[2] ≠ ...7783
3	...7531	H[3] ≠ ...7783
4	...7782	H[4] ≠ ...7783
5		H[5] = null
6		
7		
8		
9	...6284	
...		
250		

remove ...7531 : mod 251 = 1

0	...6275	
1	**..6276**	H[1] ≠ ...7531
2	...6278	H[2] ≠ ...7531
3	**..7531**	H[3] = ...7531
4	**..7782**	
5	...7535	
6	**...8033**	
7		
8		
9	...6284	
...		
250		

0	...6275
1	**...6276**
2	...6278
3	**søk**
4	**...7782**
5	...7535
6	**...8033**
7	
8	
9	...6284
...	
250	

Hash funksjoner

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)
2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$	
hash :			$\mathbf{k_i}$	$\rightarrow$ h_i

Hash funksjoner

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)
2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Key bør men trenger ikke å være injektiv:

k	:	String $\rightarrow$ int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*'en
k("*alle*")	=	
k("*anna*")	=	

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

Hash funksjoner

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)
2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Key bør men trenger ikke å være injektiv:

k	:	String $\rightarrow$ int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*'en
k("*alle*")	=	97 + 108 + 108 + 101 = 414
k("*anna*")	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

Hash funksjoner

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String $\rightarrow$ int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

Hash funksjoner

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String $\rightarrow$ int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

2. Hash

a) hvis *k1*==*k2*, så **hash**(*k1*) == **hash**(*k2*), siden **hash** er en funksjon

Hash funksjoner

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String → int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

2. Hash

a) hvis *k1*==*k2*, så **hash**(*k1*) == **hash**(*k2*), siden **hash** er en funksjon

b) for hver *k* er : **hash(*k*) = *k* mod *N* < *N***

Hash funksjoner

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String → int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

2. Hash

a) hvis *k1*==*k2*, så **hash**(*k1*) == **hash**(*k2*), siden **hash** er en funksjon

b) for hver *k* er : **hash(*k*) = *k* mod *N* < *N***

c) ingen ‘‘beste, generelle’’ hash-funksjon :

– unngå konflikter : **hash**(*k*) = 1 er ikke bra

– minimaliser gjennomsnittlig lengde for hver samling hash(k): find(*k*) = finn *k* i Container **hash**(*k*)

– alt avhenger av forventet distribusjon av data-nøkler ...

Hash funksjoner

key :	$\mathbf{n_i}$	$\rightarrow$	$\mathbf{k_i}$
hash :			$\mathbf{k_i} \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String $\rightarrow$ int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

2. Hash

a) hvis *k1*==*k2*, så **hash**(*k1*) == **hash**(*k2*), siden **hash** er en funksjon

b) for hver *k* er : **hash(*k*) = *k* mod *N* < *N***

c) ingen ‘‘beste, generelle’’ hash-funksjon :

– unngå konflikter : **hash**(*k*) = 1 er ikke bra

– minimaliser gjennomsnittlig lengde for hver samling hash(k): find(*k*) = finn *k* i Container **hash**(*k*)

– alt avhenger av forventet distribusjon av data-nøkler ...

– ***N* bør være et primtall**

20	30	40	mod 10 :	0	0	0	mod 11 :	9	8	7
23	33	43		3	3	3		1	0	10
25	35	45		5	5	5		3	2	1

Hash funksjoner

key :	n_i	$\rightarrow$	k_i
hash :			$k_i \rightarrow h_i$

1. Gitt en nøkkel *n* – et Objekt! – **konverter den til int** *k* = **k**(*n*)

2. Gitt en int-representasjon av nøkkel *k* = **k**(*n*) – finn en **hashkode** **hash**(*k*)

a) hvis **cp.isEqualTo**(*n1*,*n2*) så **hash**(**k**(*n1*)) == **hash**(**k**(*n2*)) der **cp** er den aktuelle **Comparator** for nøkler

b) **hash**(*k*) < *N* – der *N* er aktuell størrelse av tabellen

c) **hash** skal gi ‘‘jevn distribusjon‘’ – unngå kollisjoner

1. Key bør men trenger ikke å være injektiv:

k	:	String → int
k(*streng*)	=	summen av ASCII koder av alle tegn i *streng*’en
k(“*alle*”)	=	97 + 108 + 108 + 101 = 414
k(“*anna*”)	=	97 + 110 + 110 + 97 = 414

A	–	**65**
	...	
a	–	**97**
e	–	**101**
l	–	**108**
n	–	**110**

2. Hash

a) hvis *k1*==*k2*, så **hash**(*k1*) == **hash**(*k2*), siden **hash** er en funksjon

b) for hver *k* er : **hash(*k*) = *k* mod *N* < *N***

c) ingen ‘‘beste, generelle’’ hash-funksjon :

– unngå konflikter : **hash**(*k*) = 1 er ikke bra

– minimaliser gjennomsnittlig lengde for hver samling hash(k): find(*k*) = finn *k* i Container **hash**(*k*)

– alt avhenger av forventet distribusjon av data-nøkler ...

– ***N* bør være et primtall**

20	30	40	mod 10 :	0	0	0	mod 11 :	9	8	7
23	33	43		3	3	3		1	0	10
25	35	45		5	5	5		3	2	1

– **ofte brukt hash-funksjon**

hash(*k*) = (*a* *k + *b*) mod *N*, der *N* er et primtall, $a > 0$, $b \geq 0$

IV. Binært Søk

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
find(**48**)	11	19	24	32	32	48	50	55	72	99

for alle **i**, k :

- $A[k] > A[\mathbf{i}] : k > \mathbf{i}$
- $A[k] < A[\mathbf{i}] : k < \mathbf{i}$

IV. Binært Søk

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
find(**48**)	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99

for alle **i**, k :

- A[k] > A[**i**] : k > **i**
- A[k] < A[**i**] : k < **i**

IV. Binært Søk

	1	2	3	4	5	6	7	8	9	10
find(**48**)	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	**32**	48	50	55	72	99

: A[**i**]=**32** < **48** – søk til høyre

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

IV. Binært Søk

	1	2	3	4	5	6	7	8	9	10
find(**48**)	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99

: A[**i**]=32 < **48** – søk til høyre

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

IV. Binært Søk

	1	2	3	4	5	6	7	8	9	10
find(**48**)	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	**55**	72	99

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

: A[**i**]=32 < **48** – søk til høyre

: A[**i**]=**55** > **48** – søk til venstre

IV. Binært Søk

	1	2	3	4	5	6	7	8	9	10	
find(**48**)	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	48	**50**	55	72	99	: A[**i**]=**50** > **48** – søk til venstre

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

IV. Binært Søk

	1	2	3	4	5	6	7	8	9	10	
find(**48**)	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre

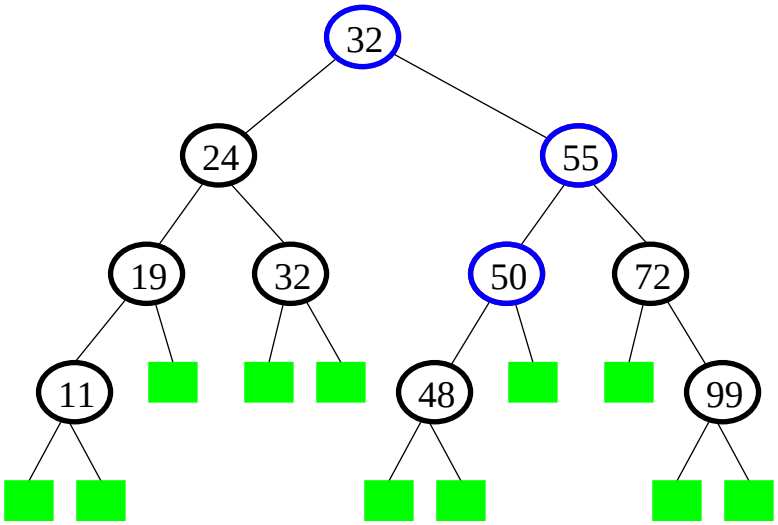
for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

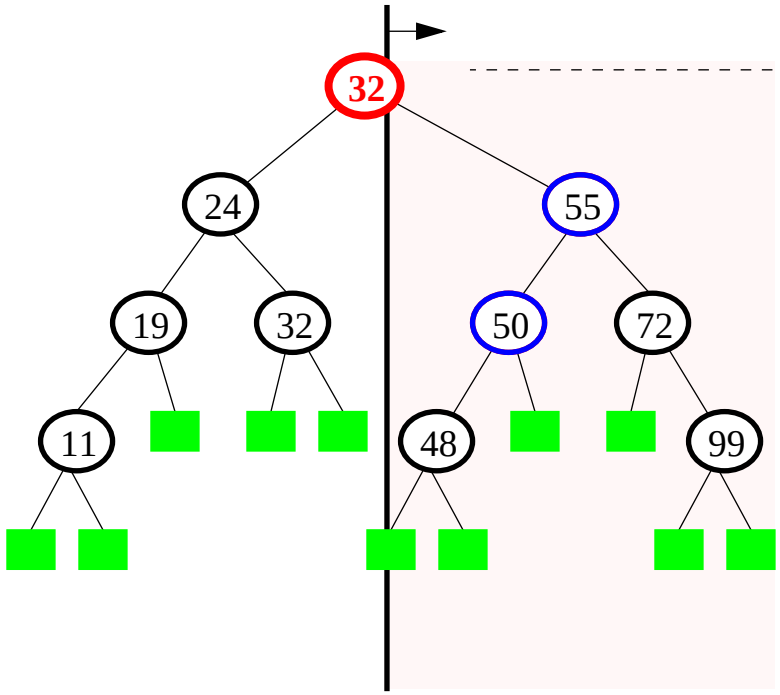
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	**48**	50	55	72	99

: A[**i**]=32 < **48** – søk til høyre

: A[**i**]=55 > **48** – søk til venstre

: A[**i**]=50 > **48** – søk til venstre

key(**i**)=**32** < **48** – søk i høyre subtre



IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

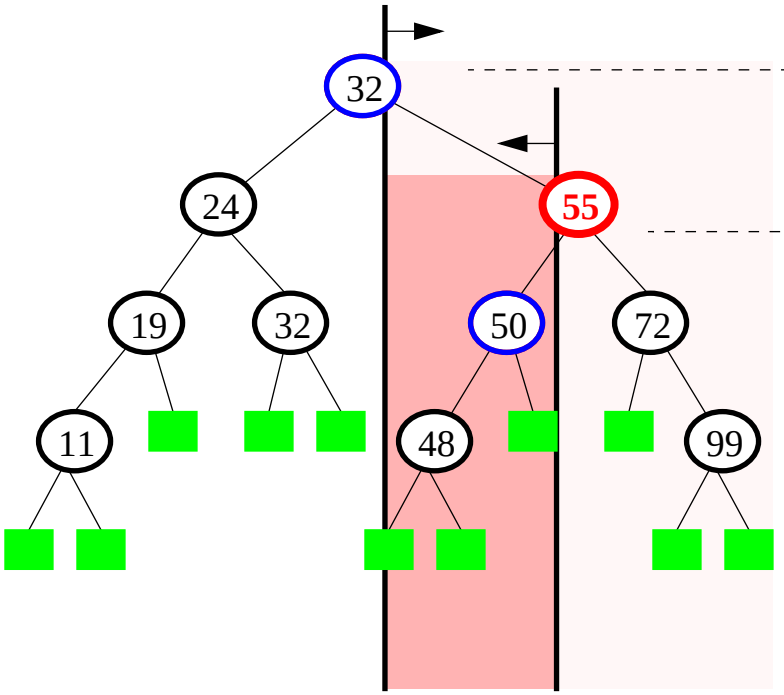
find(**48**)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	**48**	50	55	72	99

: A[**i**]=32 < **48** – søk til høyre

: A[**i**]=55 > **48** – søk til venstre

: A[**i**]=50 > **48** – søk til venstre



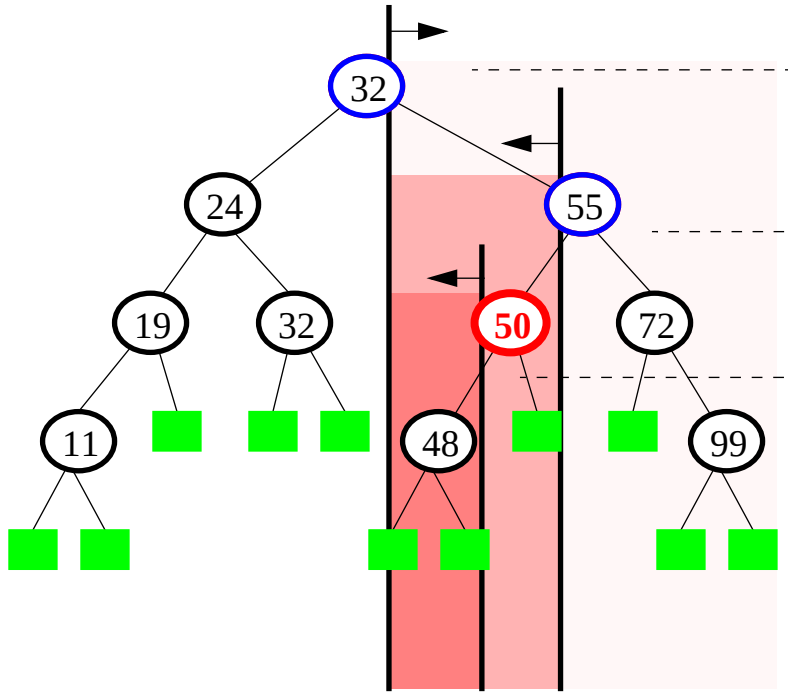
key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=**55** > **48** – søk i venstre subtre

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

	1	2	3	4	5	6	7	8	9	10	
find(**48**)	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=**50** > **48** – søk i venstre subtre

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

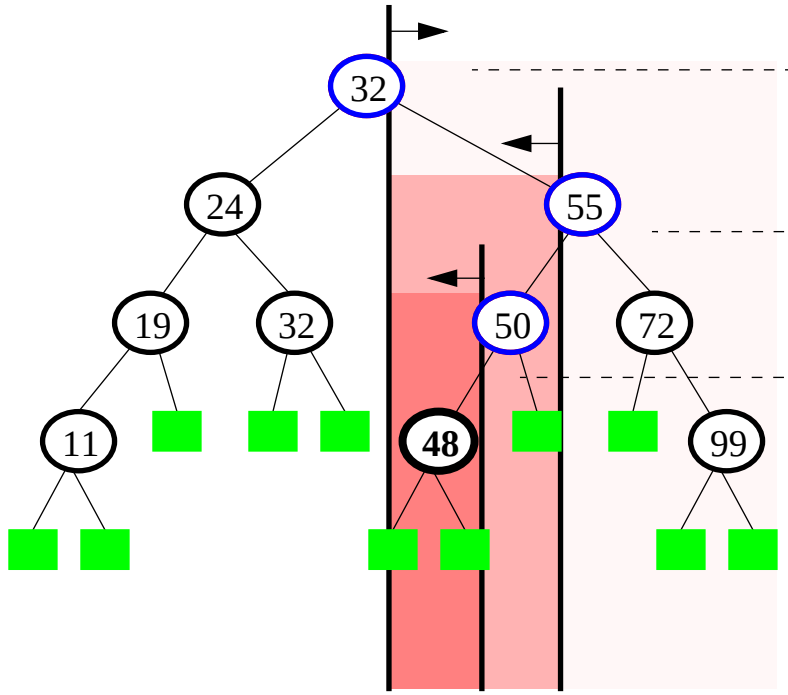
find(**48**)

	1	2	3	4	5	6	7	8	9	10
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	48	50	55	72	99
	11	19	24	32	32	**48**	50	55	72	99

: A[**i**]=32 < **48** – søk til høyre

: A[**i**]=55 > **48** – søk til venstre

: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

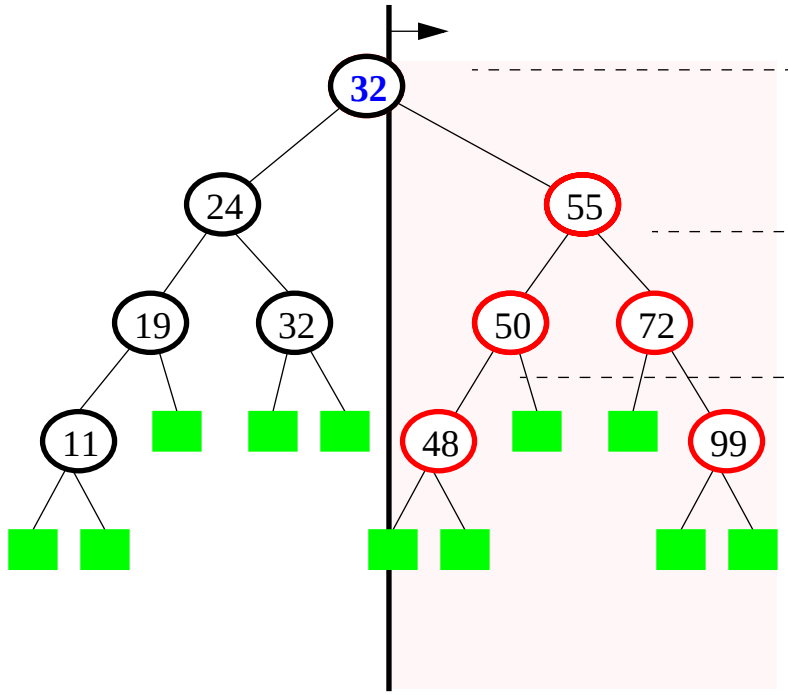
key(**i**)=50 > **48** – søk i venstre subtre

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	2	3	4	5	6	7	8	9	10	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

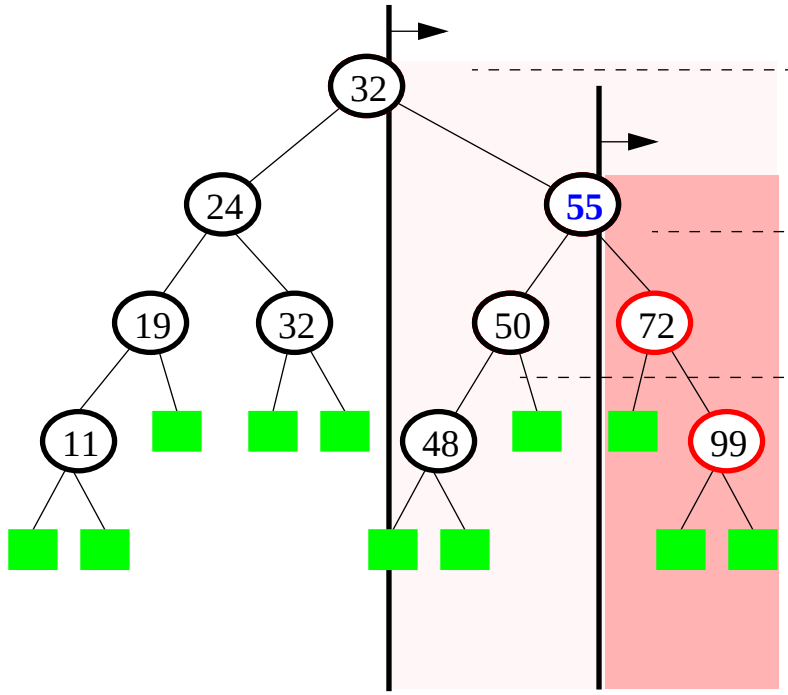
for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

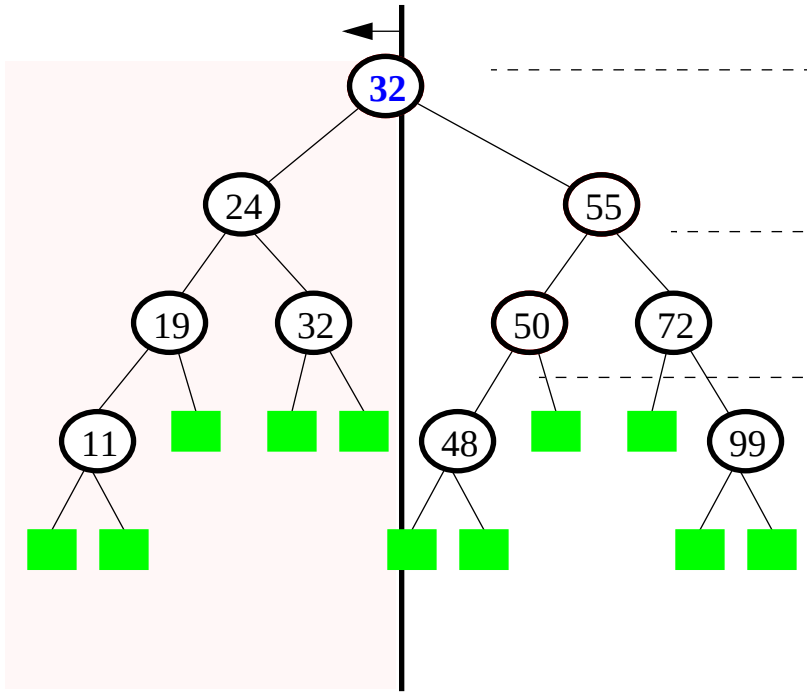
for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	2	3	4	5	6	7	8	9	10	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

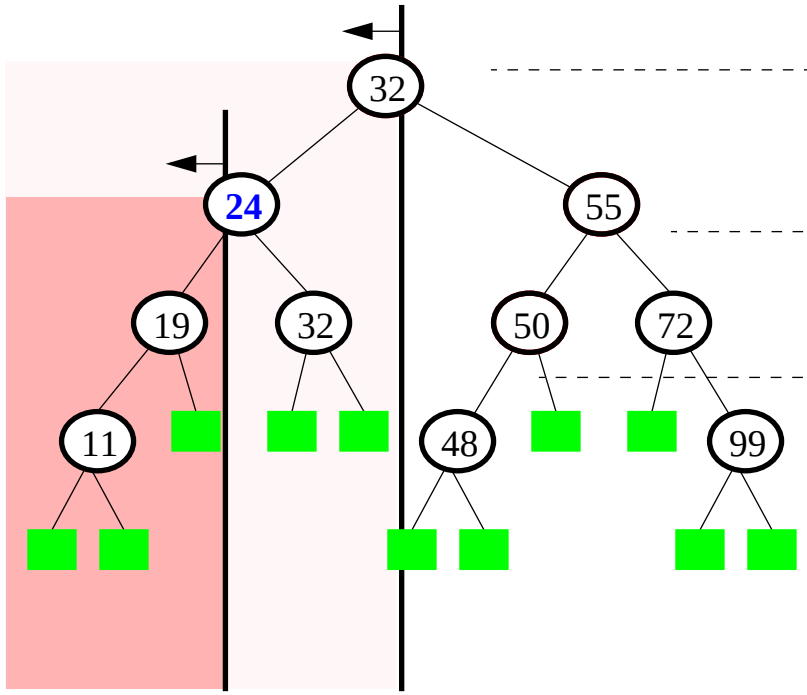
for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)
- *ko* i **venstre subtre** av **i**: key(*ko*) ≤ key(**i**)

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	2	3	4	5	6	7	8	9	10	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

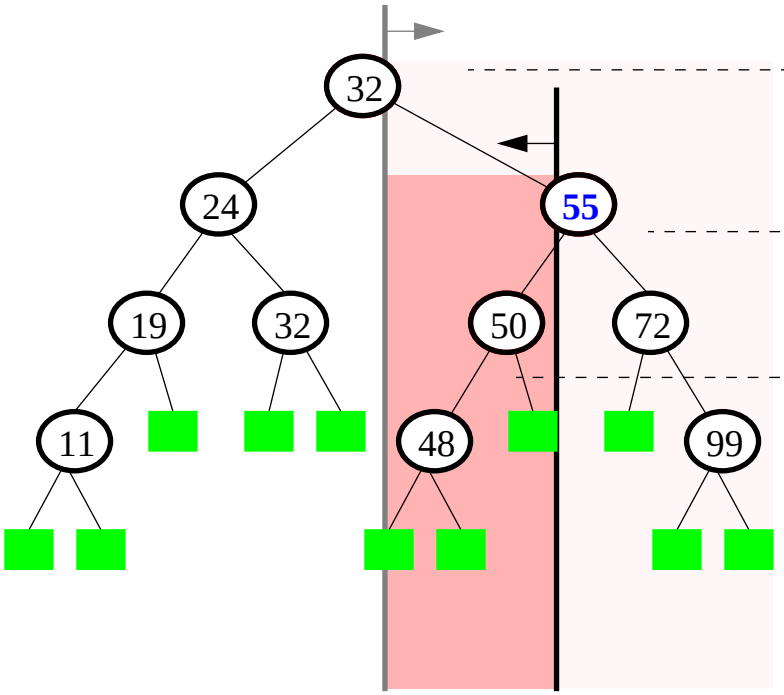
for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)
- *ko* i **venstre subtre** av **i**: key(*ko*) ≤ key(**i**)

IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
	11	19	24	32	32	48	50	55	72	99	
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
	11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
	11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)
- *ko* i **venstre subtre** av **i**: key(*ko*) ≤ key(**i**)

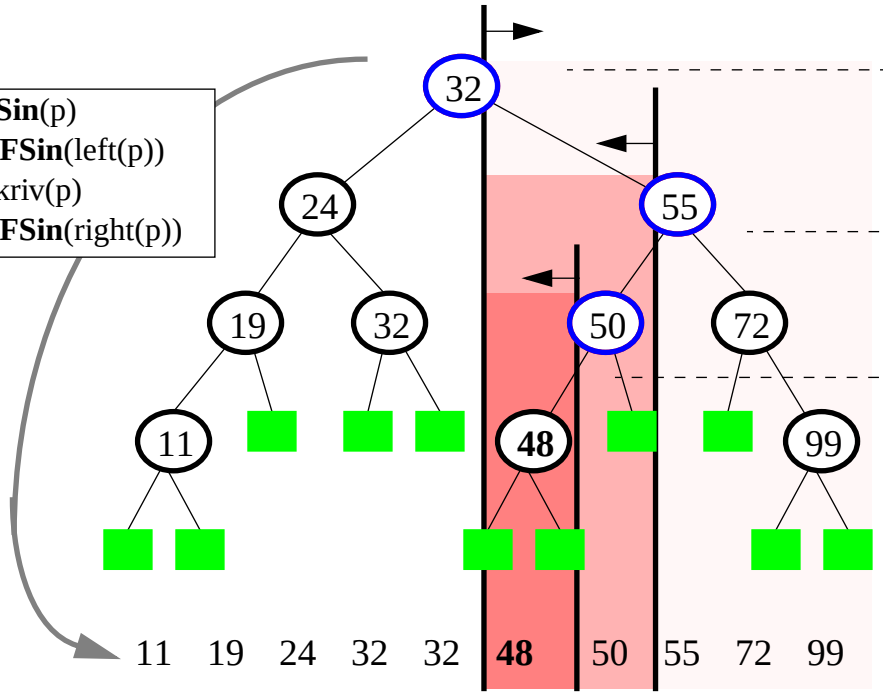
IV. Binært Søk → Binære Søketrær

for alle **i**, *k* :
- A[*k*] > A[**i**] : *k* > **i**
- A[*k*] < A[**i**] : *k* < **i**

find(**48**)

1	2	3	4	5	6	7	8	9	10	
11	19	24	32	32	48	50	55	72	99	
11	19	24	32	32	48	50	55	72	99	: A[**i**]=32 < **48** – søk til høyre
11	19	24	32	32	48	50	55	72	99	: A[**i**]=55 > **48** – søk til venstre
11	19	24	32	32	**48**	50	55	72	99	: A[**i**]=50 > **48** – søk til venstre

DFSin(p)
  DFSin(left(p))
  Skriv(p)
  DFSin(right(p))



key(**i**)=32 < **48** – søk i høyre subtre

key(**i**)=55 > **48** – søk i venstre subtre

key(**i**)=50 > **48** – søk i venstre subtre

11	19	24	32	32	**48**	50	55	72	99

for alle **i**, *ko*:
- *ko* i **høyre subtre** av **i** : key(*ko*) ≥ key(**i**)
- *ko* i **venstre subtre** av **i**: key(*ko*) ≤ key(**i**)

Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

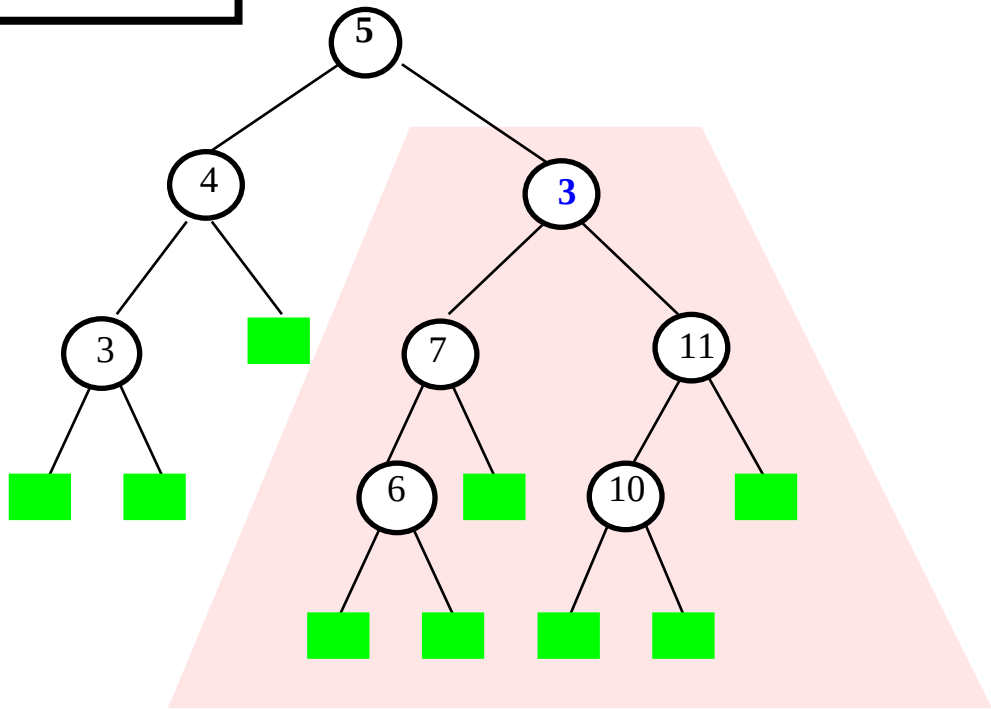
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

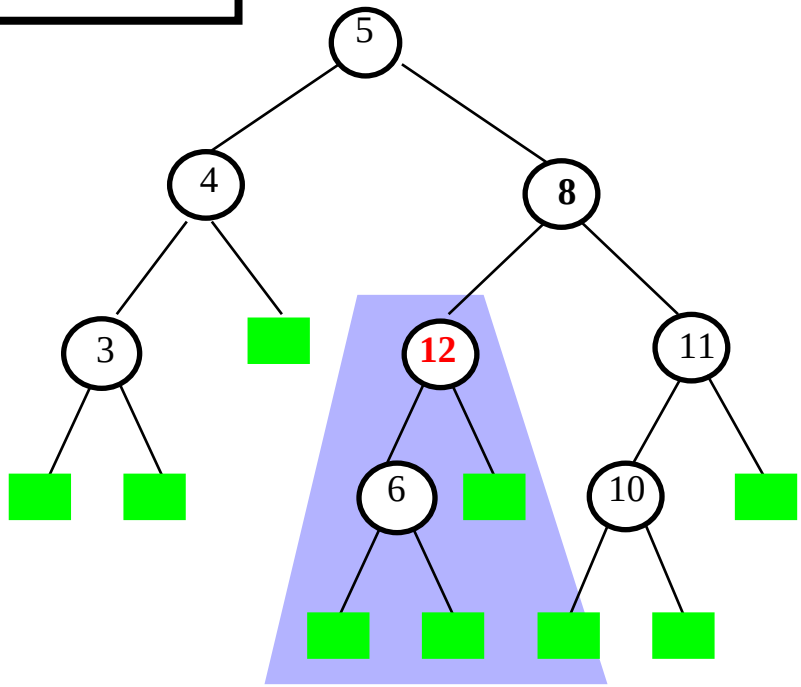
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

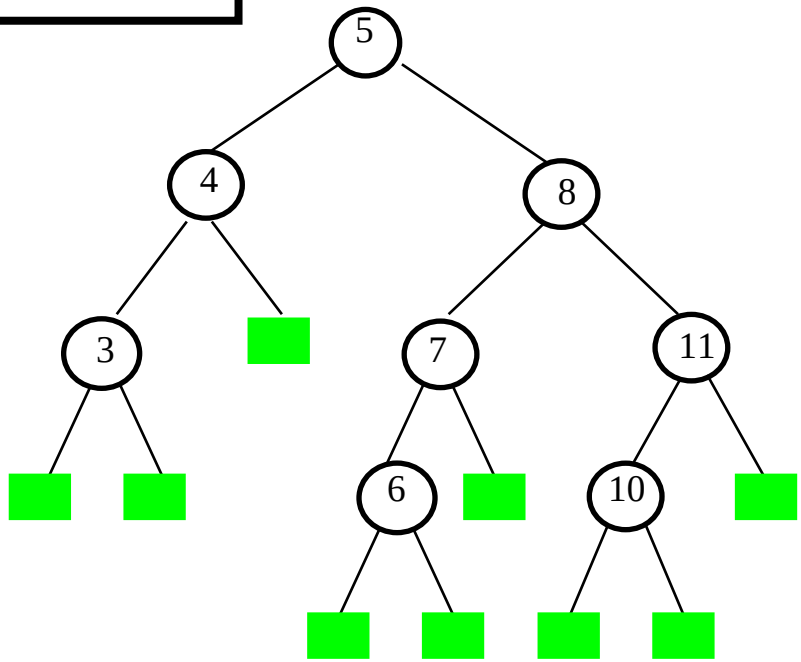
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

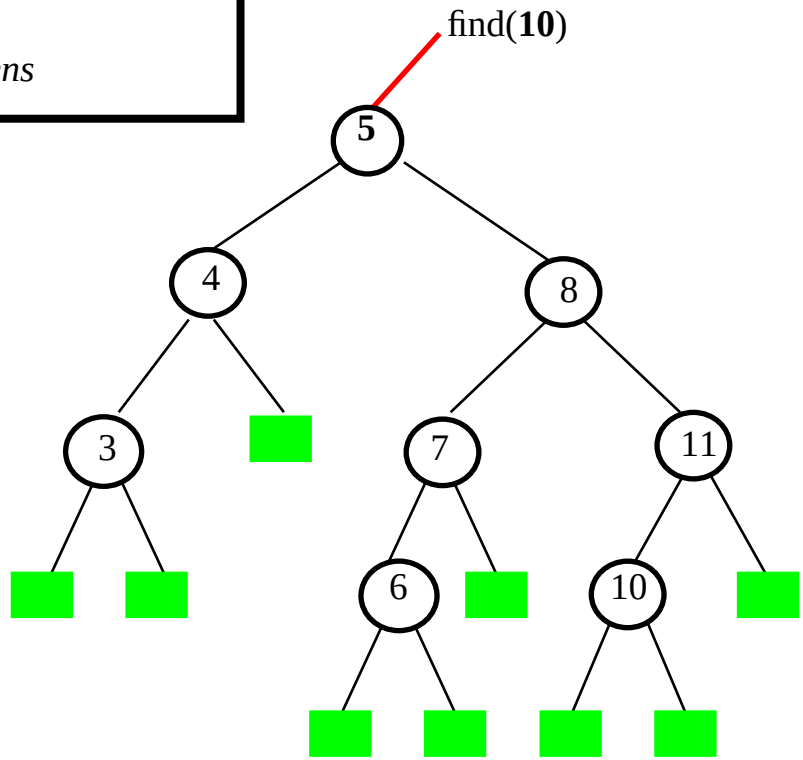
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

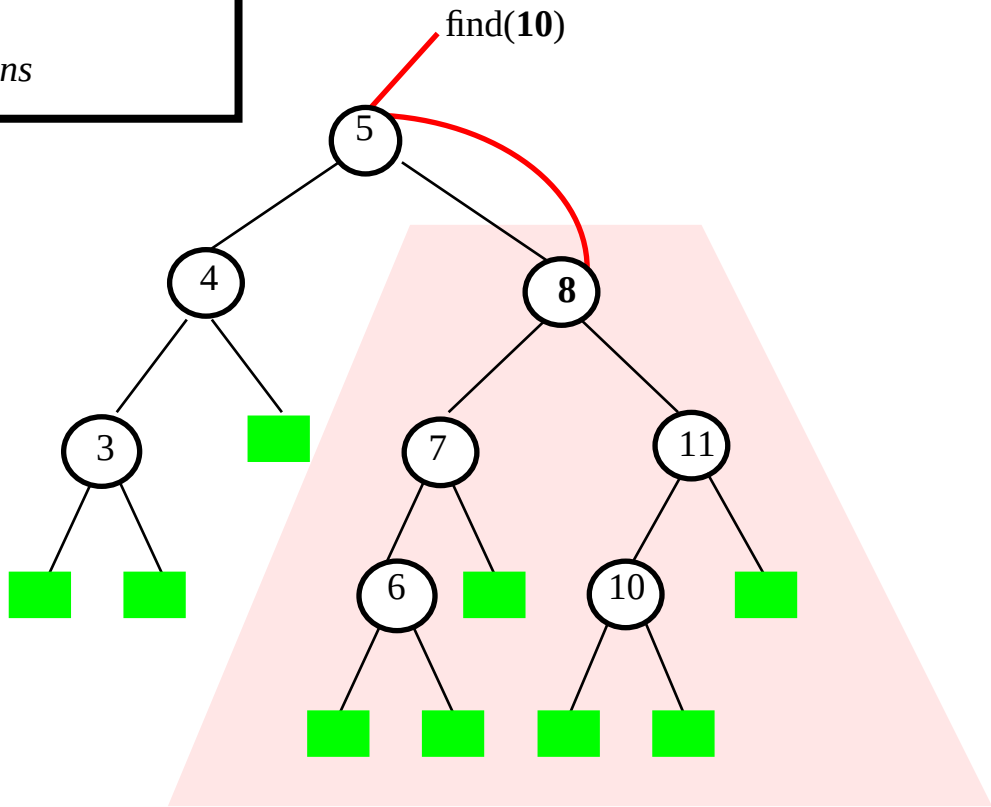
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

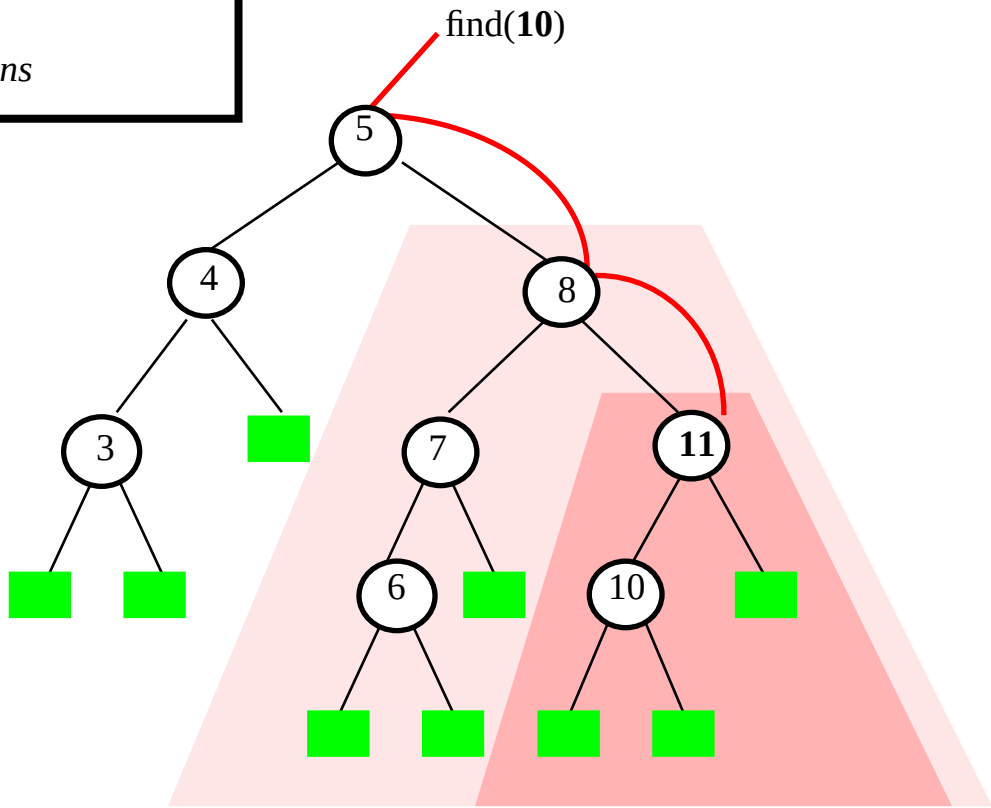
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

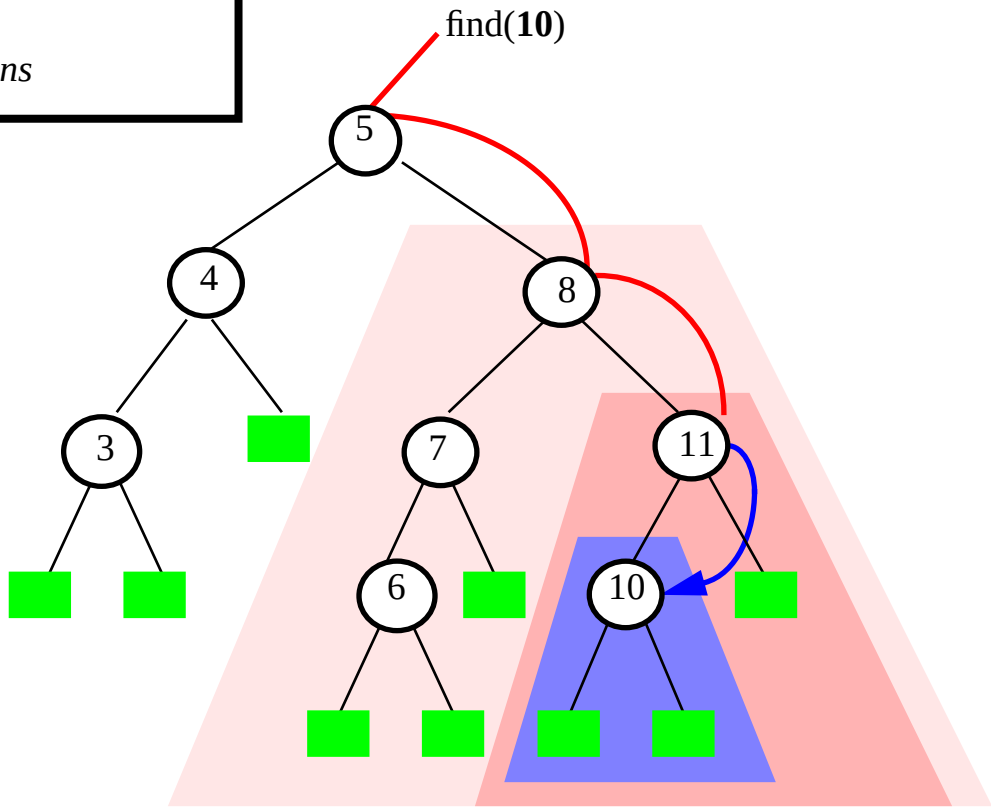
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

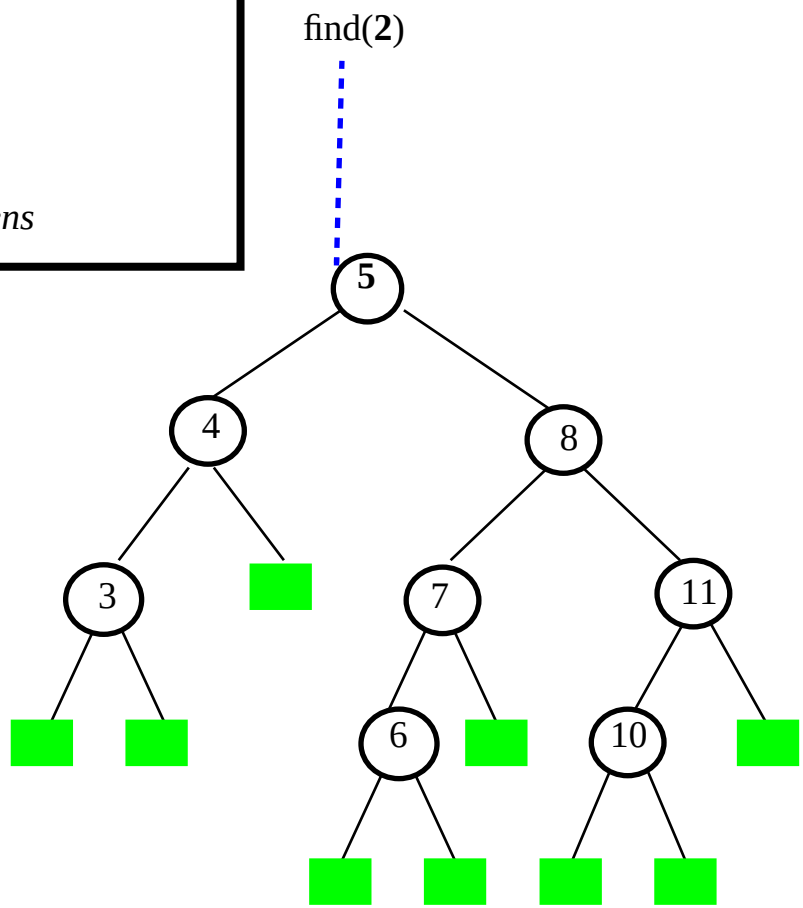
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

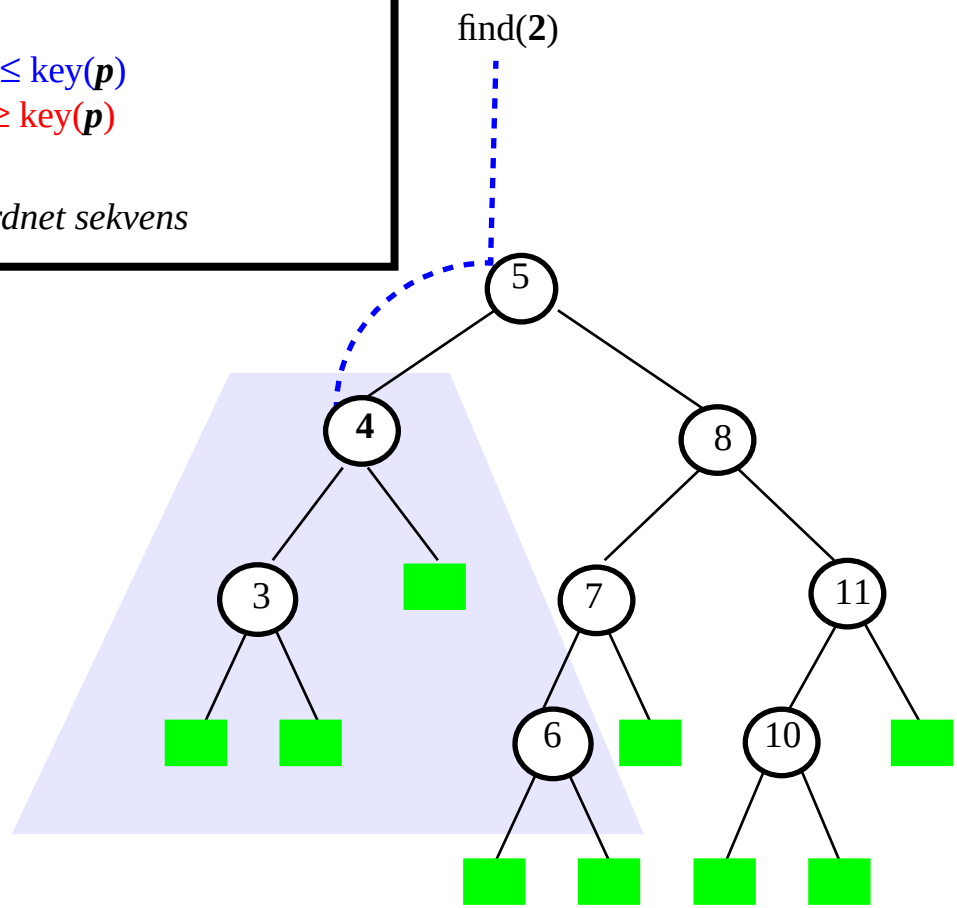
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

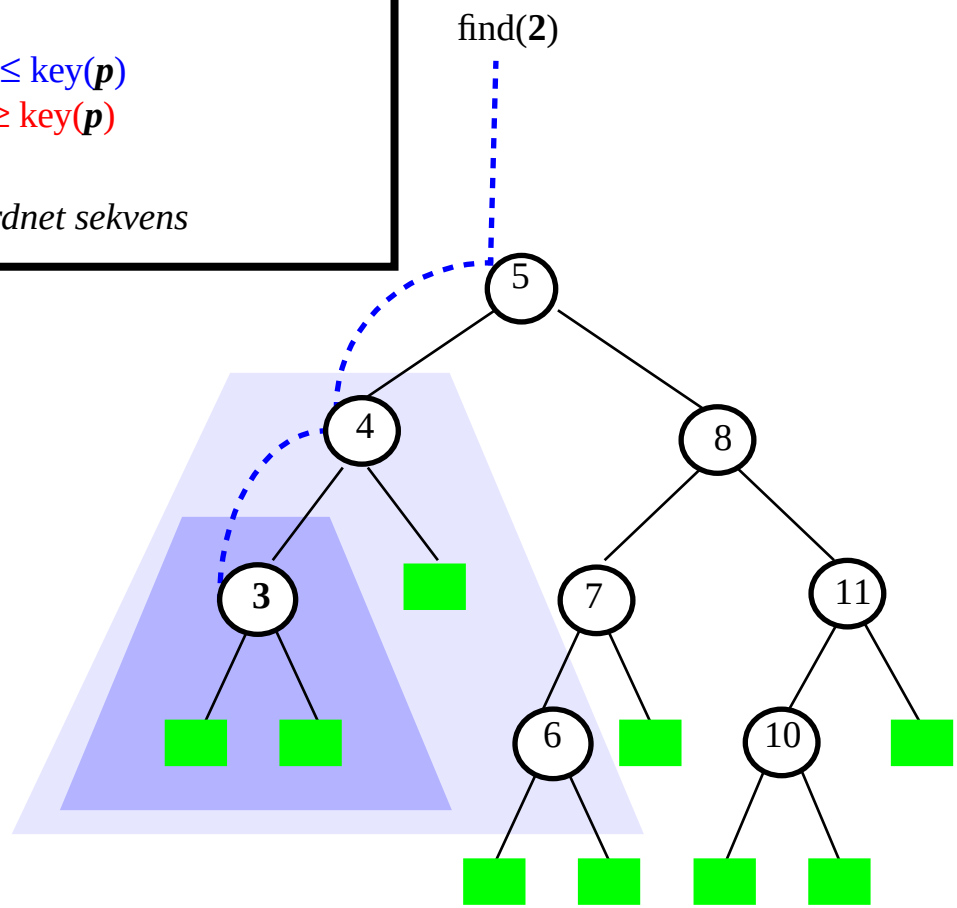
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

BST INVARIANT (“relasjonell”)

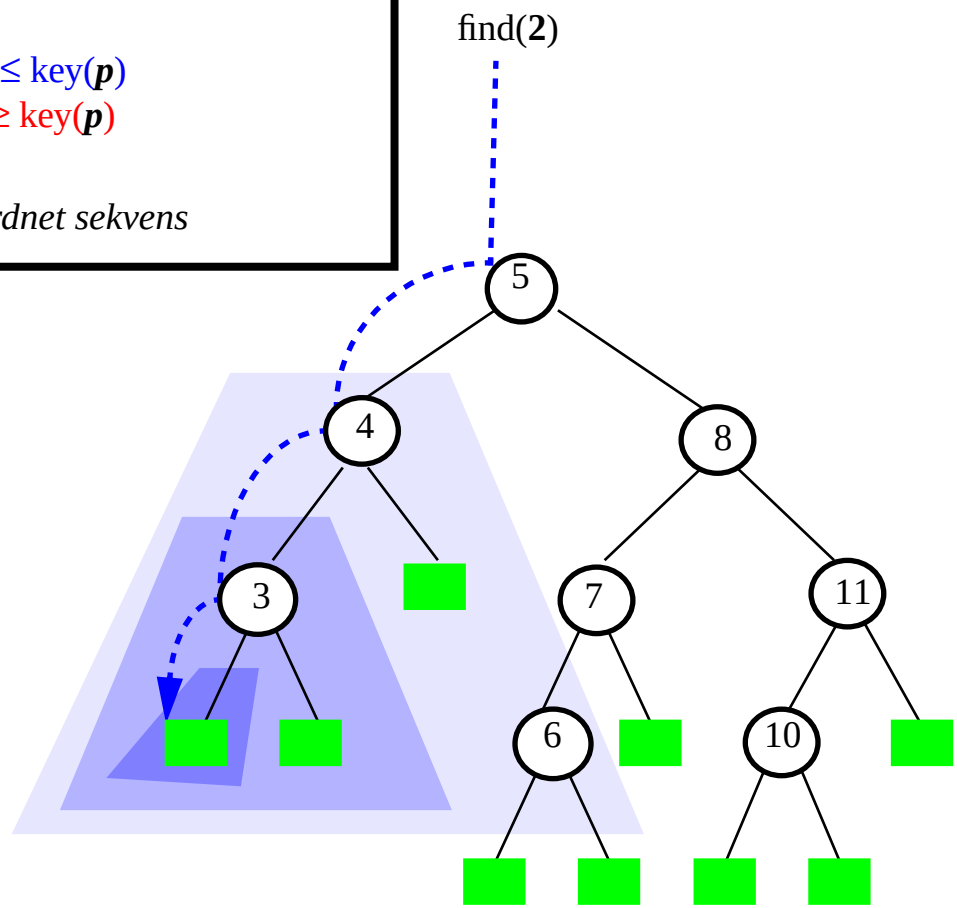
for hver node ***p*** :

for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)

for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:

DFS (inOrder) enumerering av noder gir en ordnet sekvens



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

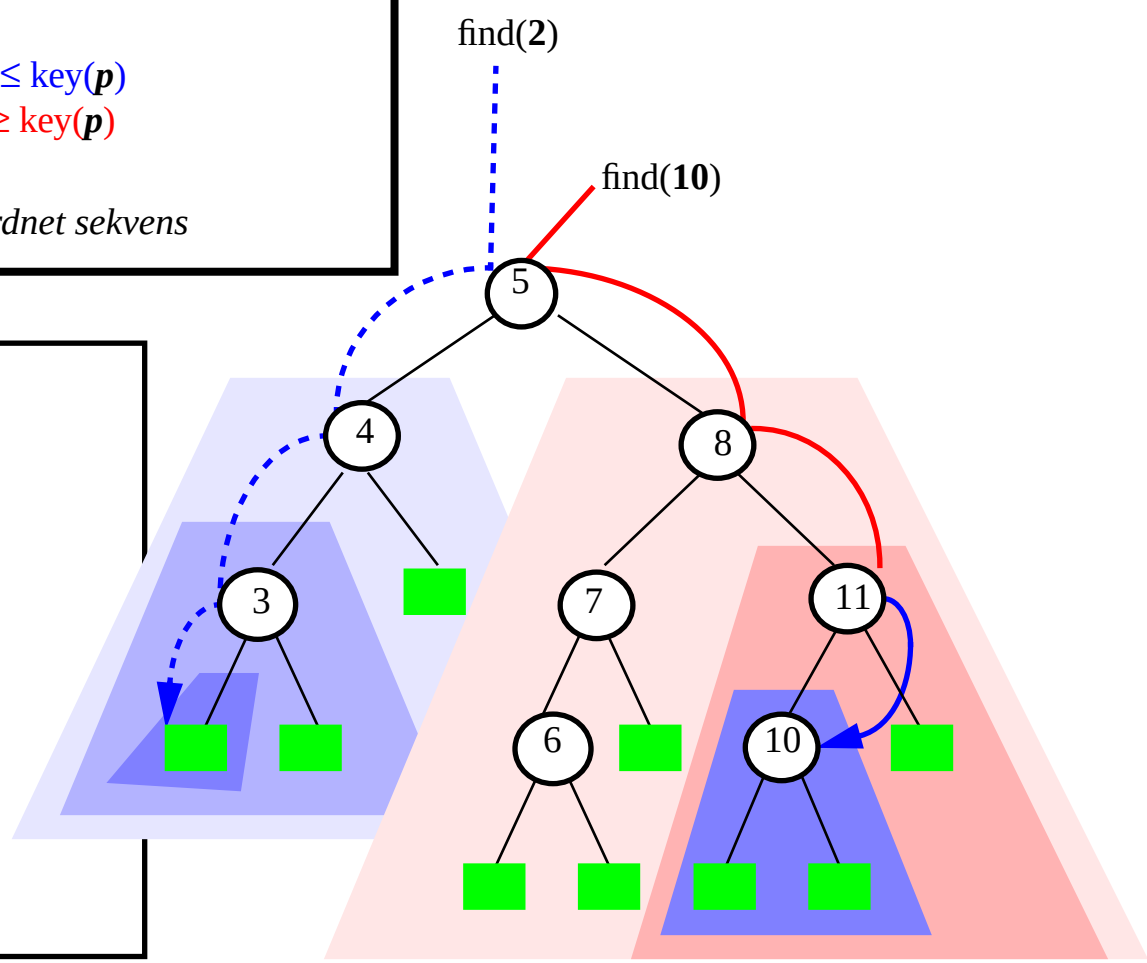
BST INVARIANT (“relasjonell”)

for hver node ***p*** :
  for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)
  for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:
  DFS (inOrder) enumerering av noder gir en ordnet sekvens

find(k) : **findPos**(new Item(k,null),T.root())

findPos (Object ko, Position p)
  if (isExternal(p)) - k finnes ikke
  else if (cp.isEqualTo(ko,key(p))) - funnet
  else if (cp.isLessThan(ko,key(p)))
    // let videre i venstre subtre
    findPos (ko,leftChild(p))
  else // cp.isGreaterThan(ko,key(p))
    // let videre i høyre subtre
    findPos (ko,rightChild(p))



Binært Søketre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

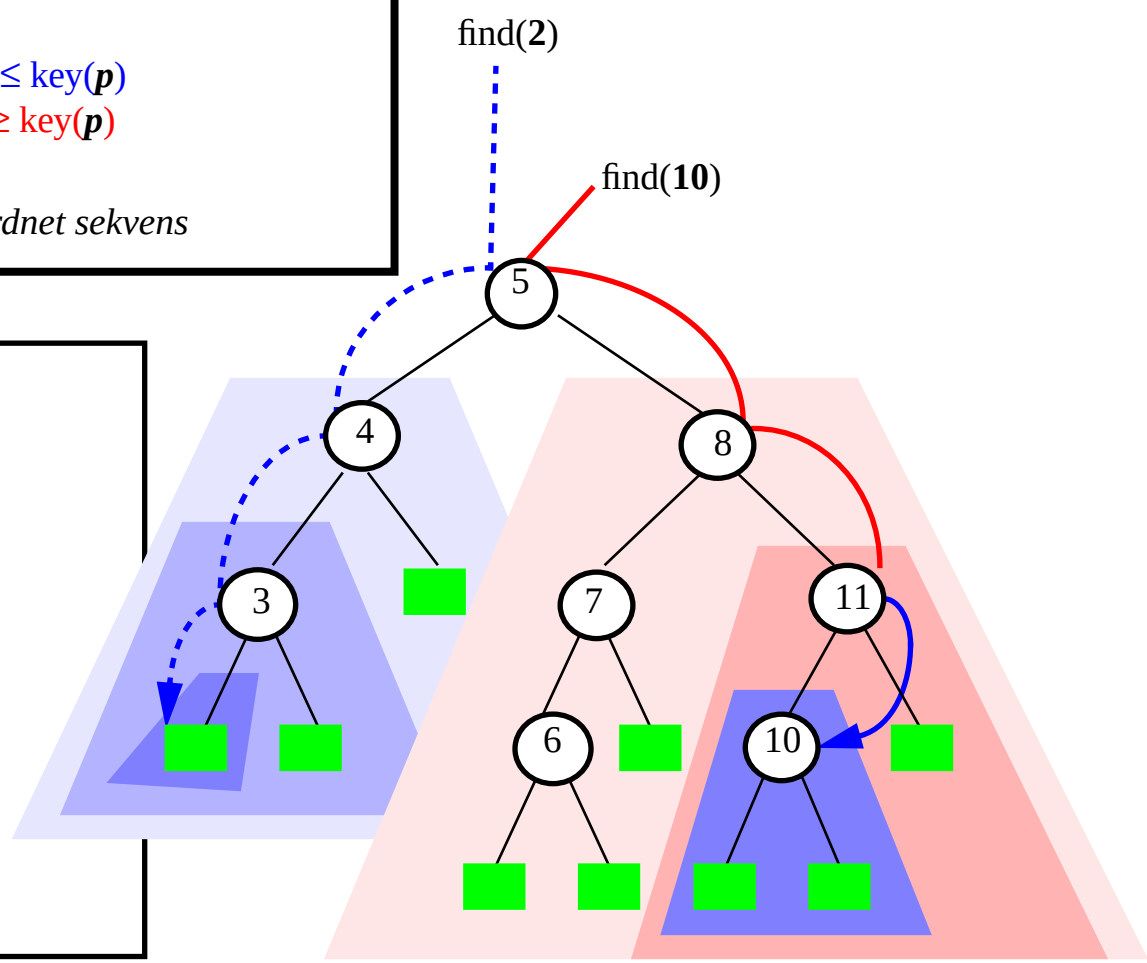
BST INVARIANT (“relasjonell”)

for hver node ***p*** :
  for enhver node *v* i ***p***’s **venstre subtre** : key(*v*) ≤ key(***p***)
  for enhver node *h* i ***p***’s **høyre subtre** : key(*h*) ≥ key(***p***)

eller:
  DFS (inOrder) enumerering av noder gir en ordnet sekvens

find(k) : **findPos**(new Item(k,null),T.root())

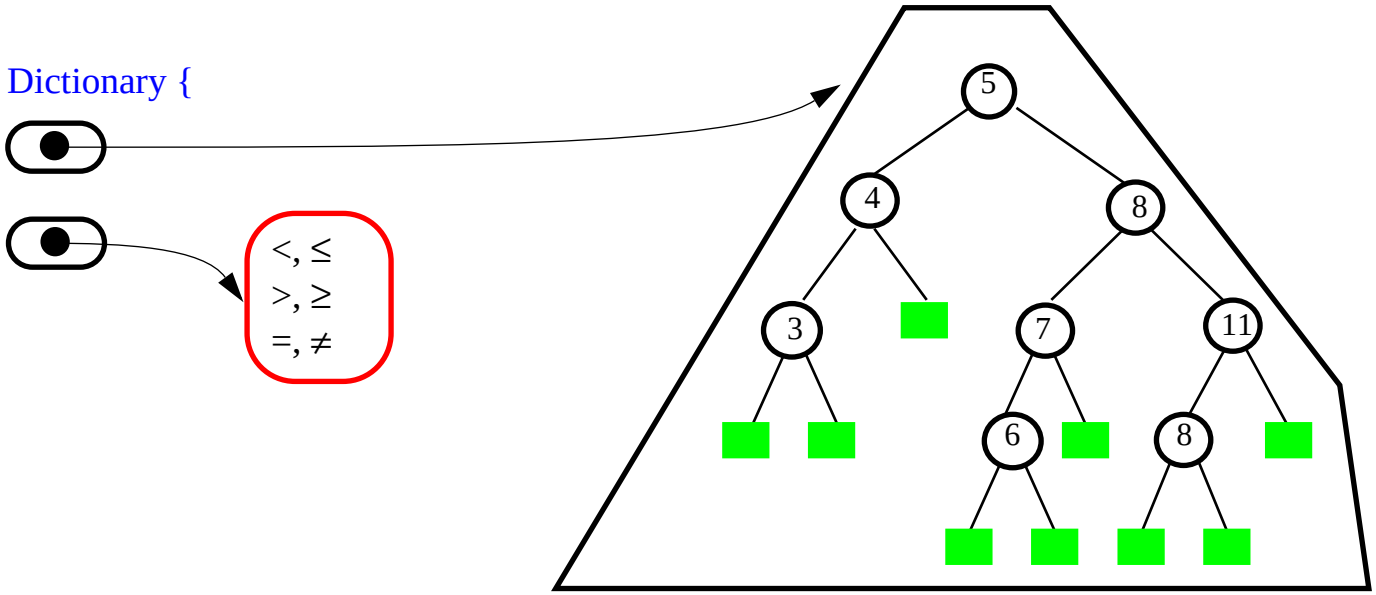
findPos (Object ko, Position p)
  if (isExternal(p)) - k finnes ikke
  else if (cp.isEqualTo(ko,key(p))) - funnet
  else if (cp.isLessThan(ko,key(p)))
    // let videre i venstre subtre
    findPos (ko,leftChild(p))
  else // cp.isGreaterThan(ko,key(p))
    // let videre i høyre subtre
    findPos (ko,rightChild(p))



= *O(* **height(T)***)*

Implementasjon av Dictionary – *med BST*

```
public class BSTDict implements Dictionary {
    protected  BinaryTree BST
    protected Comparator cp
```

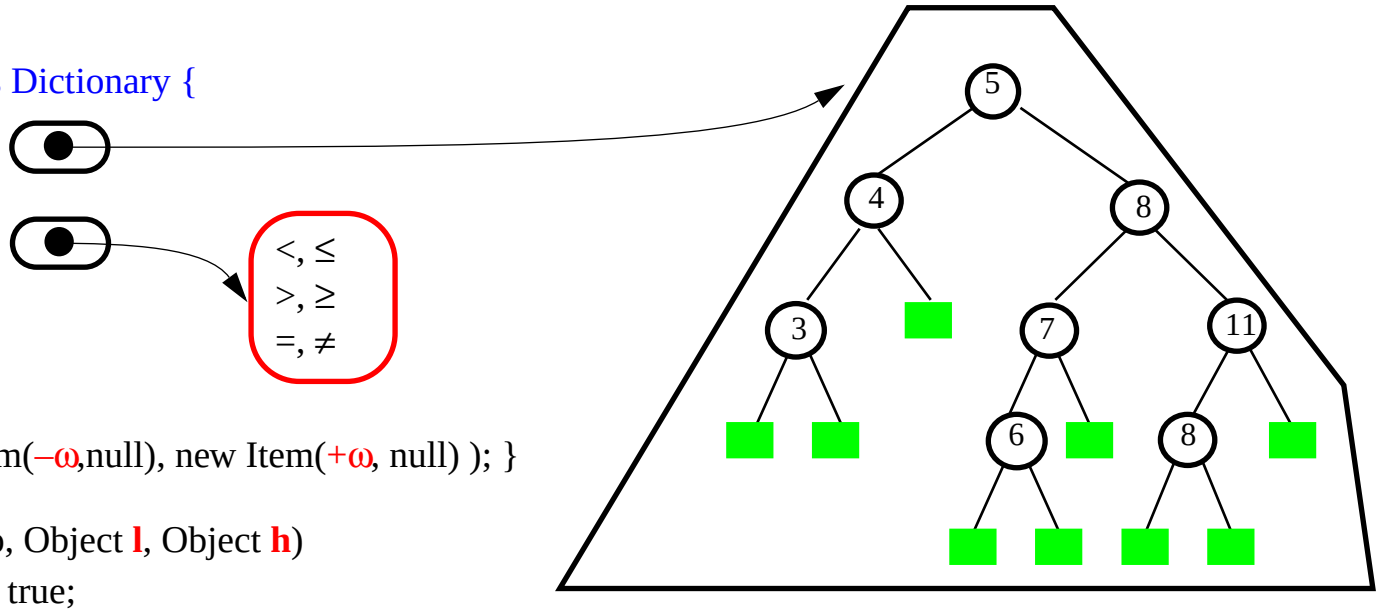


Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp

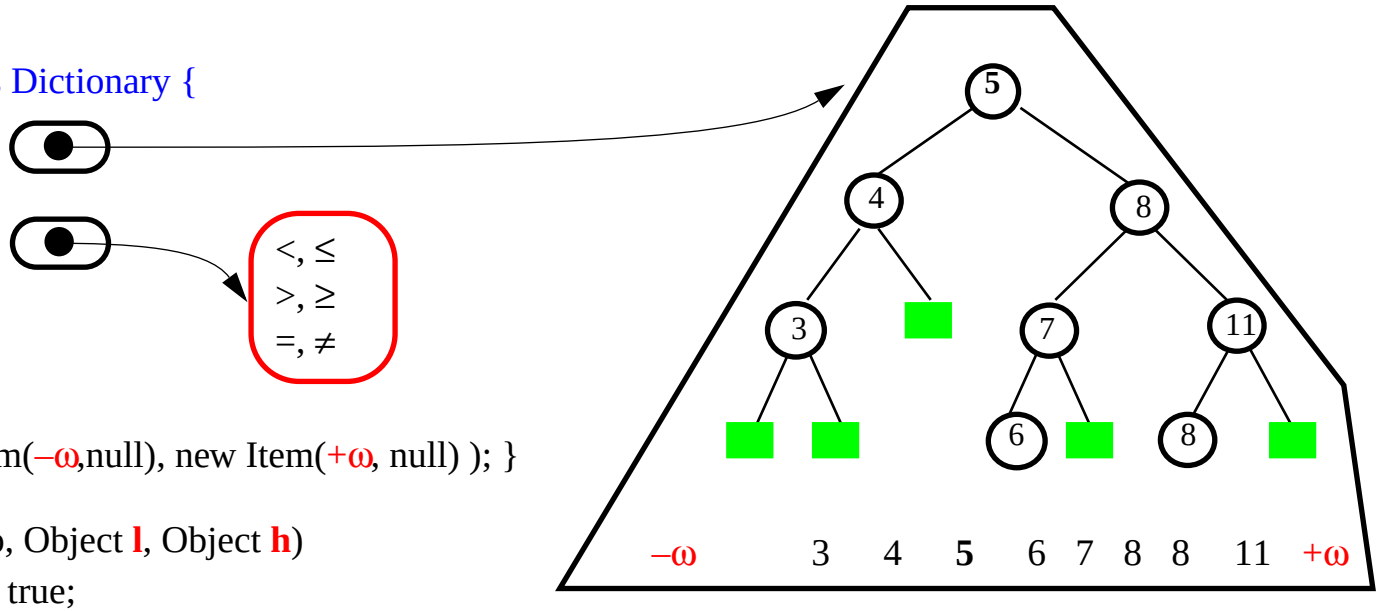


protected boolean **DI**()
{ return **DI**(BST.root(),new Item(–ω,null), new Item(+ω, null)); }

protected boolean **DI**(Position p, Object **l**, Object **h**)
{ if (BST.isExternal(p)) return true;
else if (cp.**isLessThan**(**p.element()**, **l**) || cp.**isGreaterThan**(**p.element()**, **h**)) return false;
else return (**DI**(BST.leftChild(p), **l**, **key(p)**) && **DI**(BST.rightChild(p), **key(p)**,**h**)) ; }

Implementasjon av Dictionary – *med BST*

```
public class BSTDict implements Dictionary {
   protected  BinaryTree BST

   protected Comparator cp
```



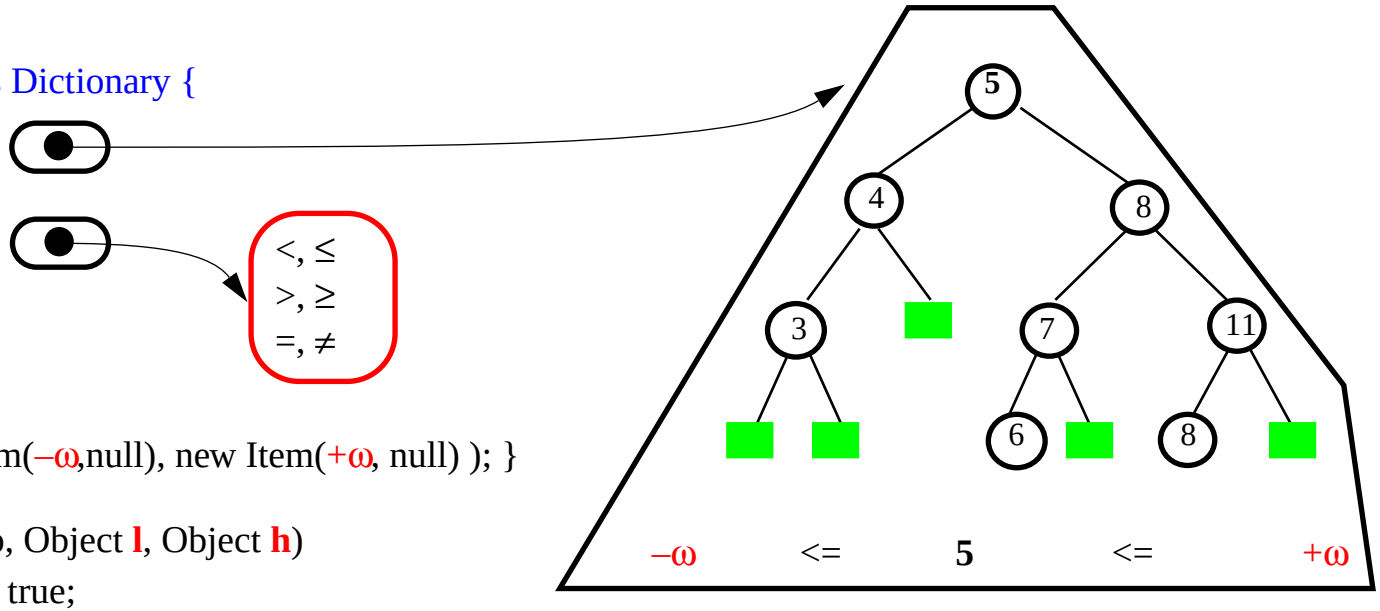

```
   protected boolean DI()
   { return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

   protected boolean DI(Position p, Object l, Object h)
   {   if (BST.isExternal(p)) return true;
       else if ( cp.isLessThan(p.element(), l)  || cp.isGreaterThan(p.element(), h) ) return false;
       else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }
```

Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp



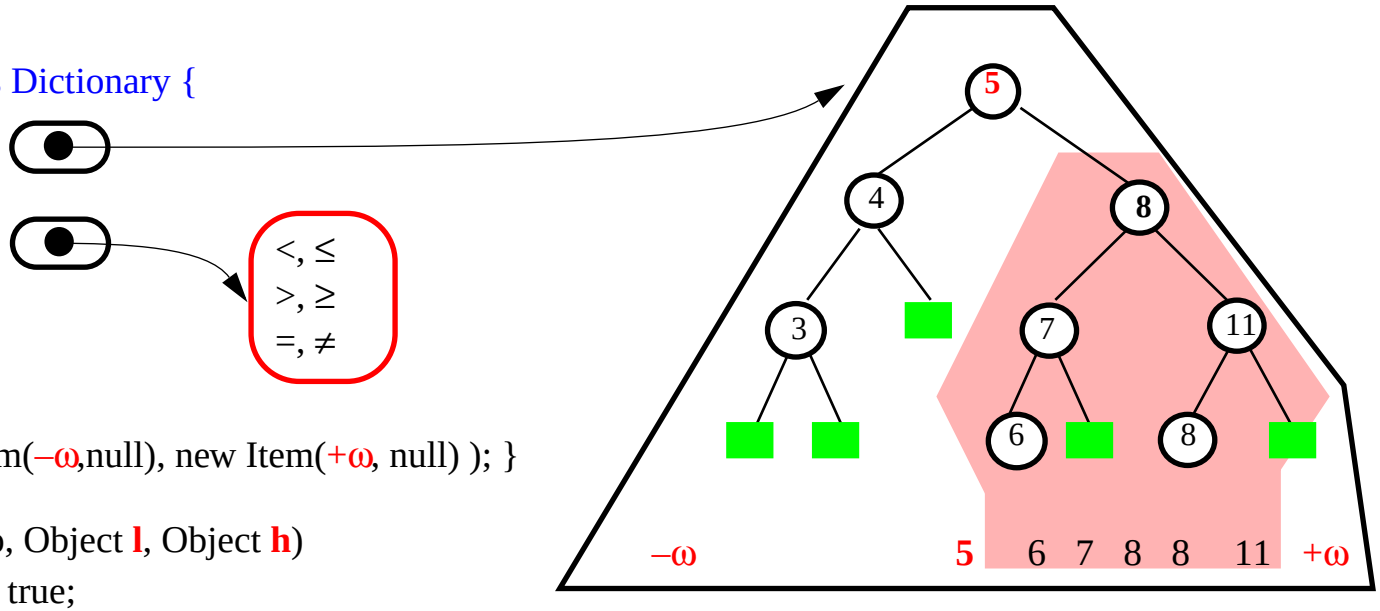

```
protected boolean DI()
{ return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

protected boolean DI(Position p, Object l, Object h)
{   if (BST.isExternal(p)) return true;
    else if ( cp.isLessThan(p.element(), l)  || cp.isGreaterThan(p.element(), h) ) return false;
    else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }
```

Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp



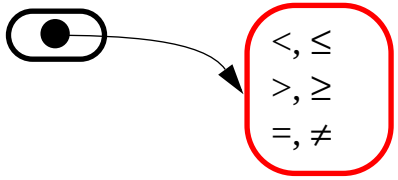

```
protected boolean DI()
{ return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

protected boolean DI(Position p, Object l, Object h)
{   if (BST.isExternal(p)) return true;
    else if ( cp.isLessThan(p.element(), l)  || cp.isGreaterThan(p.element(), h) ) return false;
    else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }
```

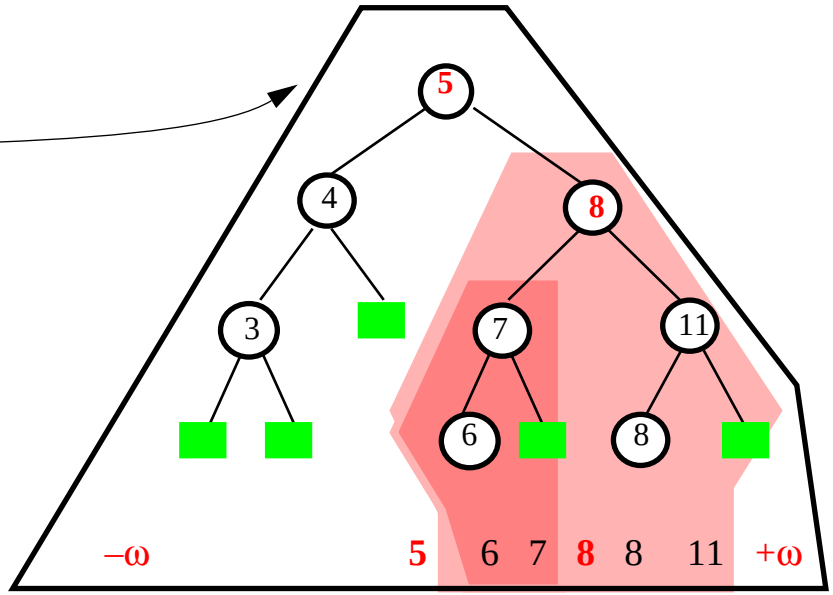
Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp





protected boolean **DI**()
{ return **DI**(BST.root(),new Item(−ω,null), new Item(+ω, null)); }

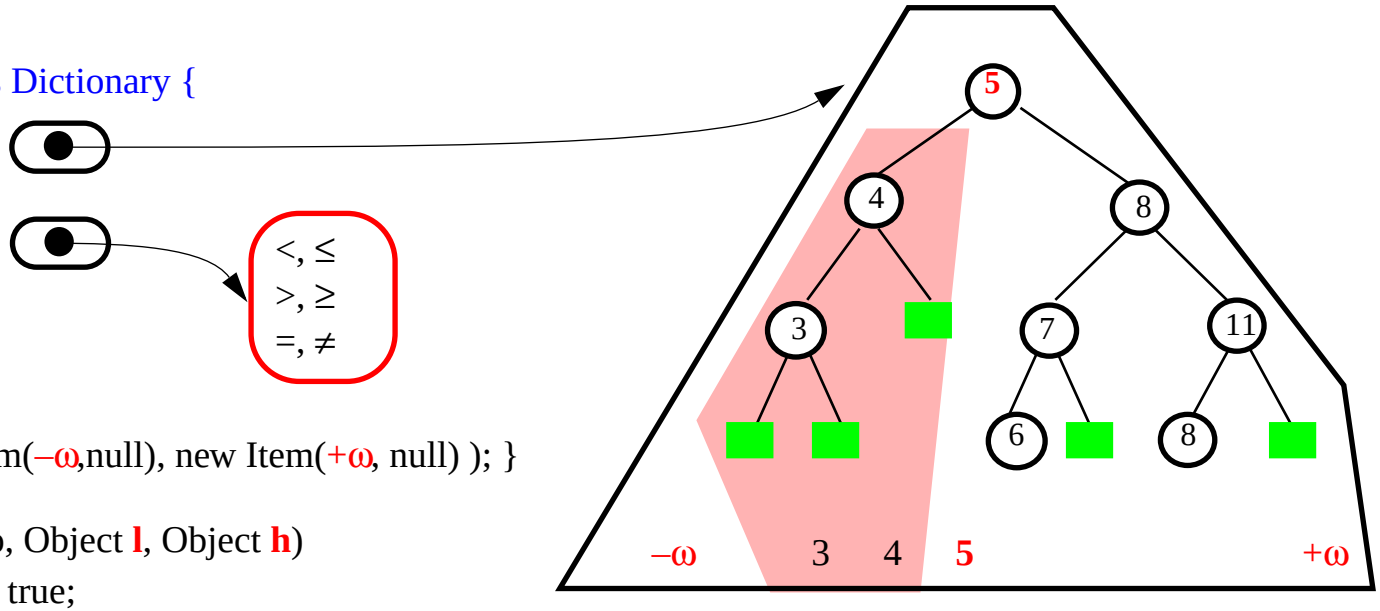
protected boolean **DI**(Position p, Object **l**, Object **h**)
{ if (BST.isExternal(p)) return true;
else if (cp.**isLessThan**(**p.element()**, **l**) || cp.**isGreaterThan**(**p.element()**, **h**)) return false;
else return (**DI**(BST.**leftChild**(p), **l**, **key(p)**) && **DI**(BST.rightChild(p), **key(p)**,**h**)) ; }

Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp



protected boolean **DI**()
{ return **DI**(BST.root(),new Item(–ω,null), new Item(+ω, null)); }

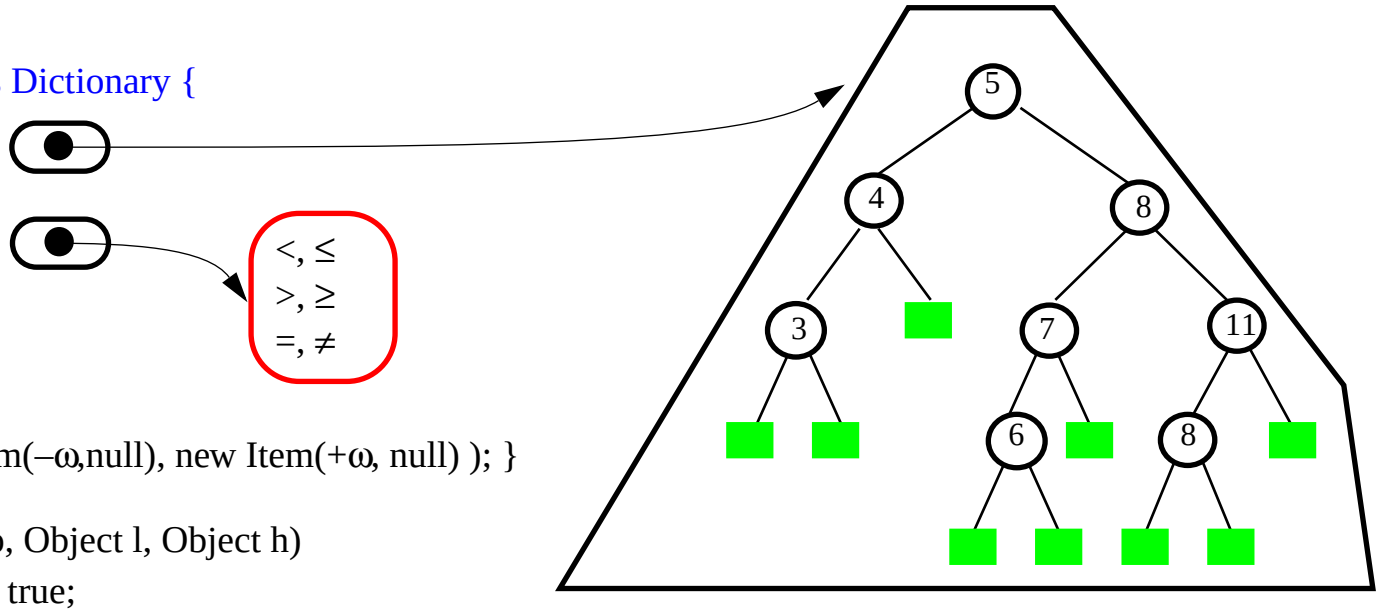
protected boolean **DI**(Position p, Object **l**, Object **h**)
{ if (BST.isExternal(p)) return true;
else if (cp.**isLessThan**(**p.element()**, **l**) || cp.**isGreaterThan**(**p.element()**, **h**)) return false;
else return (**DI**(BST.**leftChild**(p), **l**, **key(p)**) && **DI**(BST.rightChild(p), **key(p)**,**h**)) ; }

Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

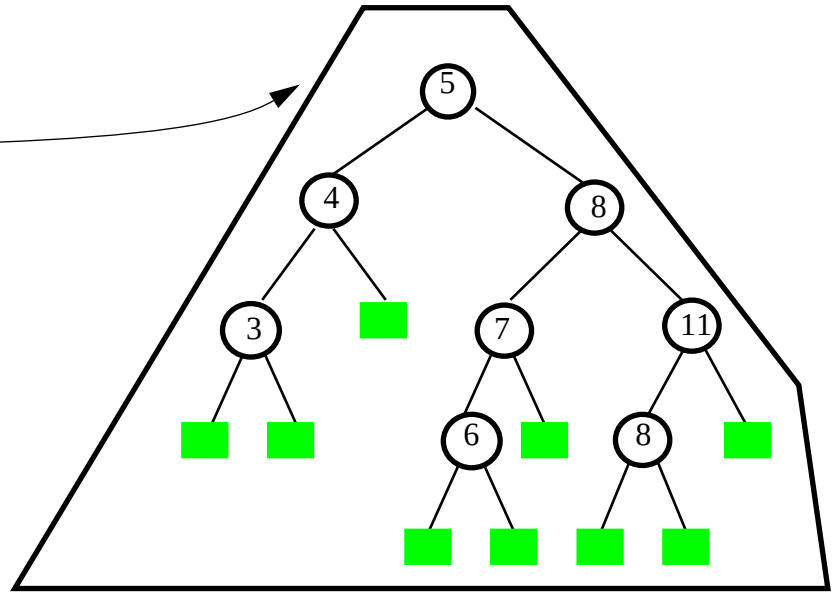
protected **BinaryTree** BST

protected **Comparator** cp

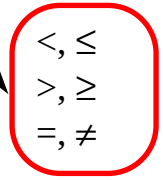



```
protected boolean DI()
{ return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

protected boolean DI(Position p, Object l, Object h)
{   if (BST.isExternal(p)) return true;
    else if ( cp.isLessThan(p.element(), l) || cp.isGreaterThan(p.element(), h) ) return false;
    else return   (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }

protected Object findPos(Object k) { return findPos(k,BST.root());}

public Object find(Object k)
    { Position p = findPos(k) ;
        if ( BST.isExternal(p) ) throw new NoSuchKey("...");
        return ((Item) p.element()) . element() ; }
```

Implementasjon av Dictionary – *med BST*




```
public class BSTDict implements Dictionary {
   protected  BinaryTree BST

   protected Comparator cp
```




```
  protected boolean DI()
  { return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

  protected boolean DI(Position p, Object l, Object h)
  {   if (BST.isExternal(p)) return true;
      else if ( cp.isLessThan(p.element(), l) || cp.isGreaterThan(p.element(), h) ) return false;
      else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }

  protected Object findPos(Object k) { return findPos(k,BST.root());}

  public Object find(Object k)
      { Position p = findPos(k) ;
          if ( BST.isExternal(p) ) throw new NoSuchKey("...");
          return ((Item) p.element()) . element() ; }
```

```
Position findPos(Object k, Position p)
  { ... }
```

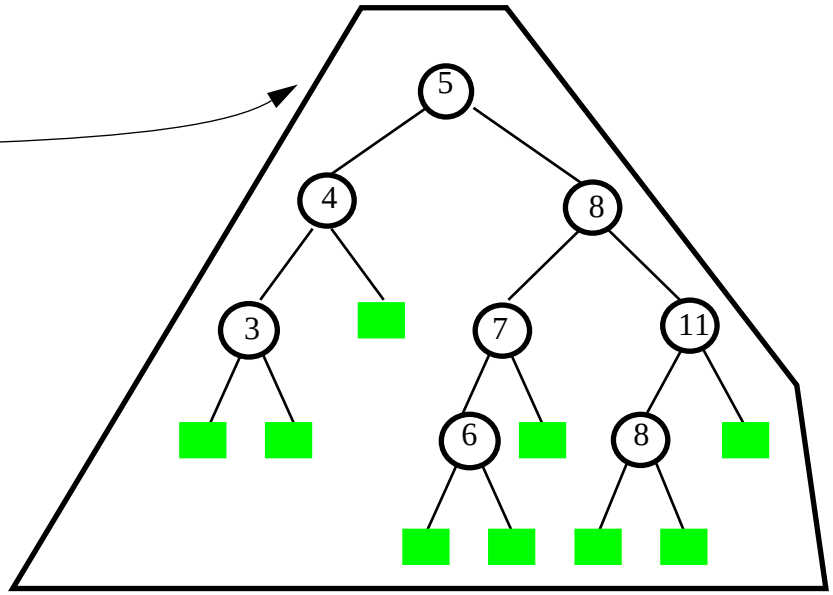
Implementasjon av Dictionary – *med BST*

public class BSTDict implements Dictionary {

protected **BinaryTree** BST

protected **Comparator** cp

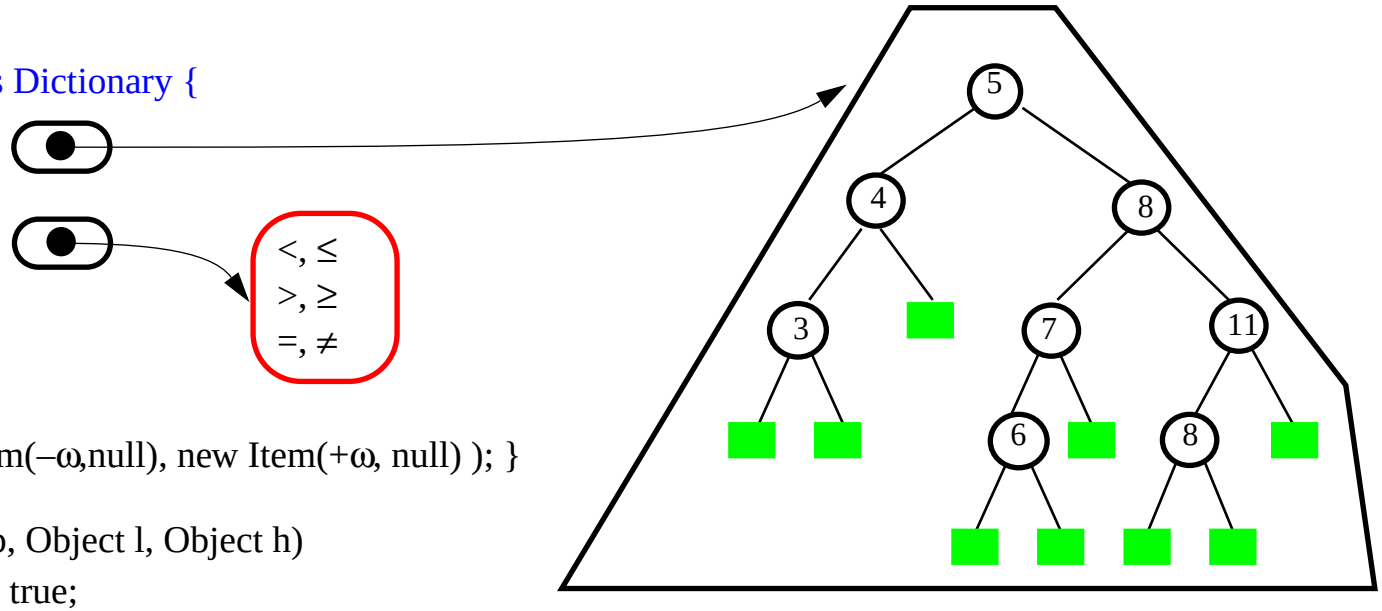
<, ≤
>, ≥
=, ≠




```
protected boolean DI()
{ return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

protected boolean DI(Position p, Object l, Object h)
{   if (BST.isExternal(p)) return true;
    else if ( cp.isLessThan(p.element(), l) || cp.isGreaterThan(p.element(), h) ) return false;
    else return   (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }

protected Object findPos(Object k) { return findPos(k,BST.root());}

public Object find(Object k)
    { Position p = findPos(k) ;
        if ( BST.isExternal(p) ) throw new NoSuchKey("...");
        return ((Item) p.element()) . element() ; }

public ObjectIterator findAll(Object k)
    { velg en implementasjon av Iterator E= new Enum();
        findAll(k, BST.root(), E) ;
        return E;  }
```

Position **findPos**(Object k, Position p)
{ ... }

Implementasjon av Dictionary – *med BST*




```
public class BSTDict implements Dictionary {
   protected  BinaryTree BST

   protected Comparator cp
```

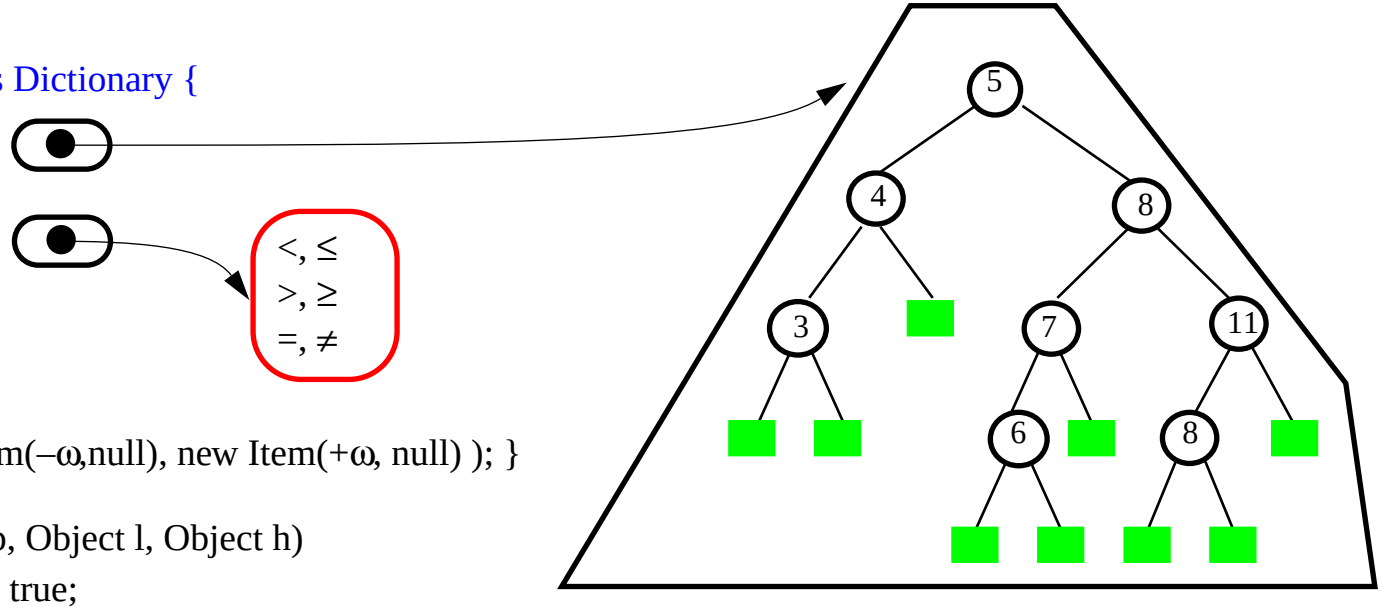
<, ≤
>, ≥
=, ≠

```
   protected boolean DI()
   { return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

   protected boolean DI(Position p, Object l, Object h)
   {   if (BST.isExternal(p)) return true;
       else if ( cp.isLessThan(p.element(), l) || cp.isGreaterThan(p.element(), h) ) return false;
       else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }

   protected Object findPos(Object k) { return findPos(k,BST.root());}

   public Object find(Object k)
       { Position p = findPos(k) ;
           if ( BST.isExternal(p) ) throw new NoSuchKey("...");
           return ((Item) p.element()) . element() ; }

   public ObjectIterator findAll(Object k)
       { velg en implementasjon av Iterator E= new Enum();
           findAll(k, BST.root(), E) ;
           return E;  }
```

```
Position findPos(Object k, Position p)
  { ... }
```

```
void findAll(Object k, Position p, Iter E) {
   f= findPos(k,p);
   if (p != null && BST.isInternal(p))
      E.add( ((Item)p.element()).element() );
      findAll(k, BST.leftChild(p), E);
      findAll(k, BST.rightChild(p), E);
}
```

Implementasjon av Dictionary – *med BST*




```
public class BSTDict implements Dictionary {
   protected  BinaryTree BST

   protected Comparator cp
```

<, ≤
>, ≥
=, ≠

```
  protected boolean DI()
  { return DI( BST.root(),new Item(–ω,null), new Item(+ω, null) ); }

  protected boolean DI(Position p, Object l, Object h)
  {   if (BST.isExternal(p)) return true;
      else if ( cp.isLessThan(p.element(), l) || cp.isGreaterThan(p.element(), h) ) return false;
      else return    (DI(BST.leftChild(p), l, key(p)) && DI(BST.rightChild(p), key(p),h)) ; }

  protected Object findPos(Object k) { return findPos(k,BST.root());}

  public Object find(Object k)
      { Position p = findPos(k) ;
          if ( BST.isExternal(p) ) throw new NoSuchKey("...");
          return ((Item) p.element()) . element() ; }

  public ObjectIterator findAll(Object k)
      { velg en implementasjon av Iterator E= new Enum();
          findAll(k, BST.root(), E) ;
          return E;  }
  . . . insert, remove
```

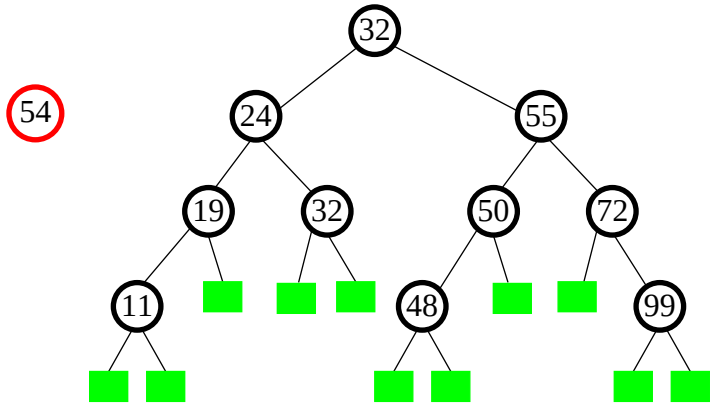
```
Position findPos(Object k, Position p)
  { ... }
```

```
void findAll(Object k, Position p, Iter E) {
   f= findPos(k,p);
   if (p != null && BST.isInternal(p))
      E.add( ((Item)p.element()).element() );
      findAll(k, BST.leftChild(p), E);
      findAll(k, BST.rightChild(p), E);
}
```

insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

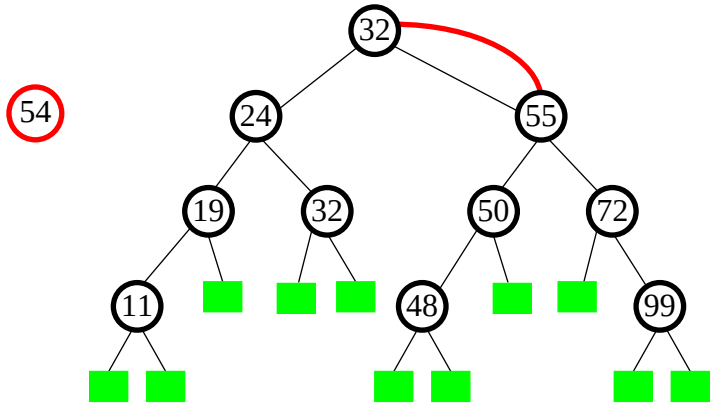
Position **p** = **findPos**(k, v)



insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

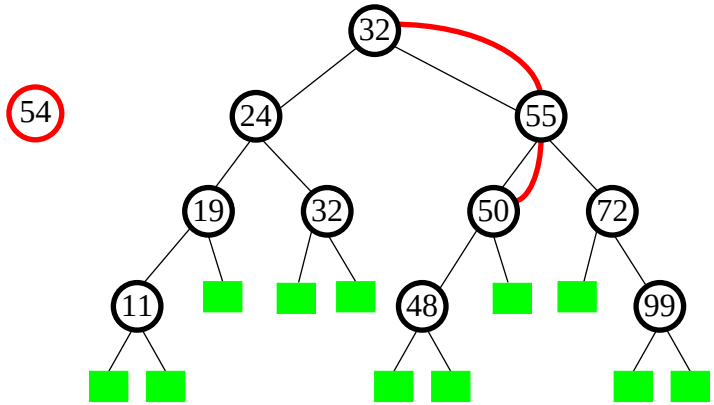
Position **p** = **findPos**(k, v)



insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

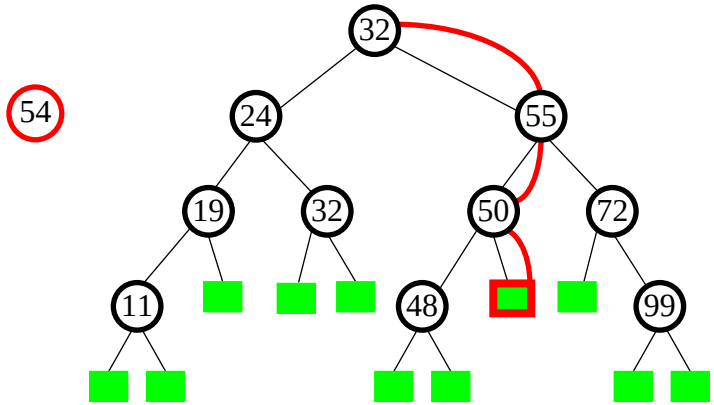
Position **p** = **findPos**(k, v)



insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

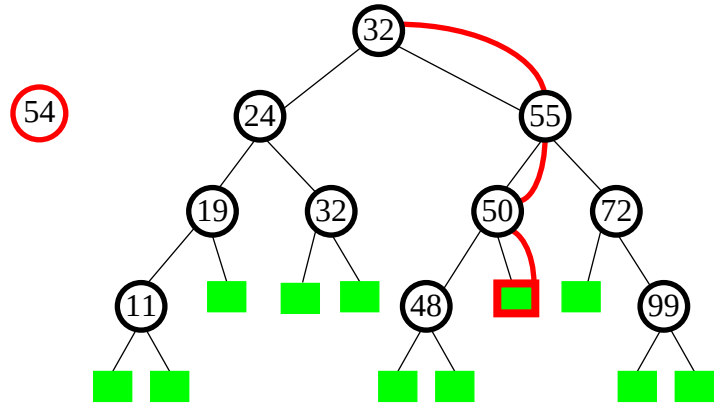


insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

sikrer
BST INVARIANT

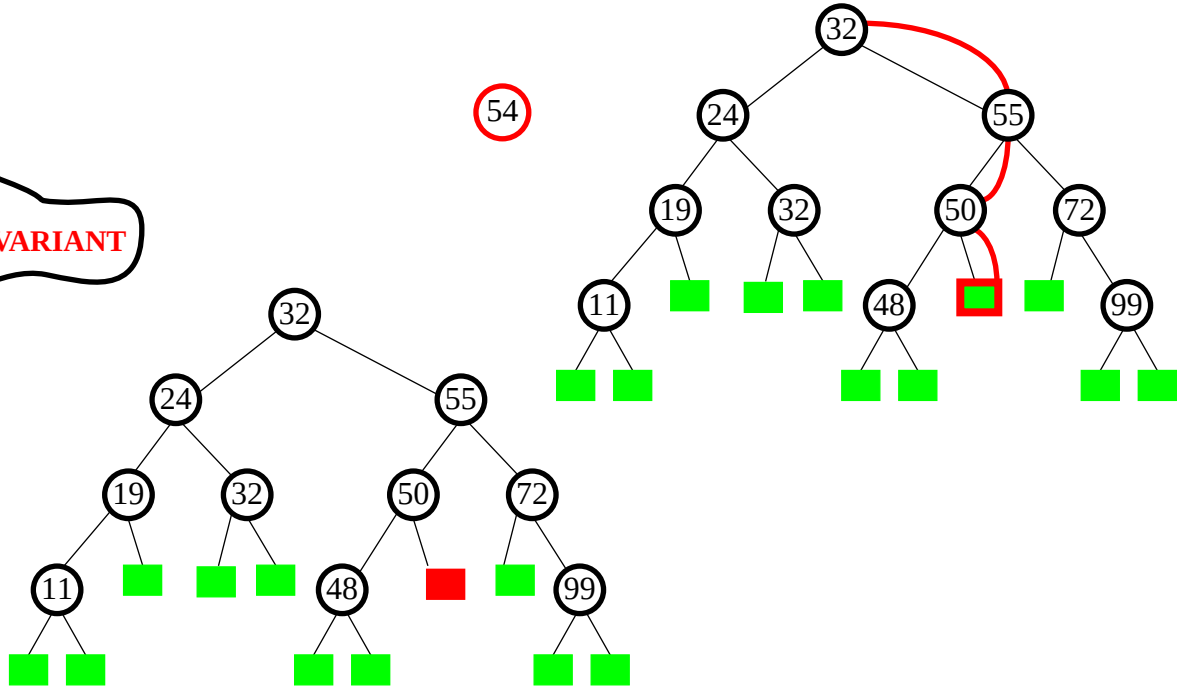


insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

sikrer
BST INVARIANT



if (BST.**isExternal**(**p**))

ny Intern node: expandExternal(**p**)

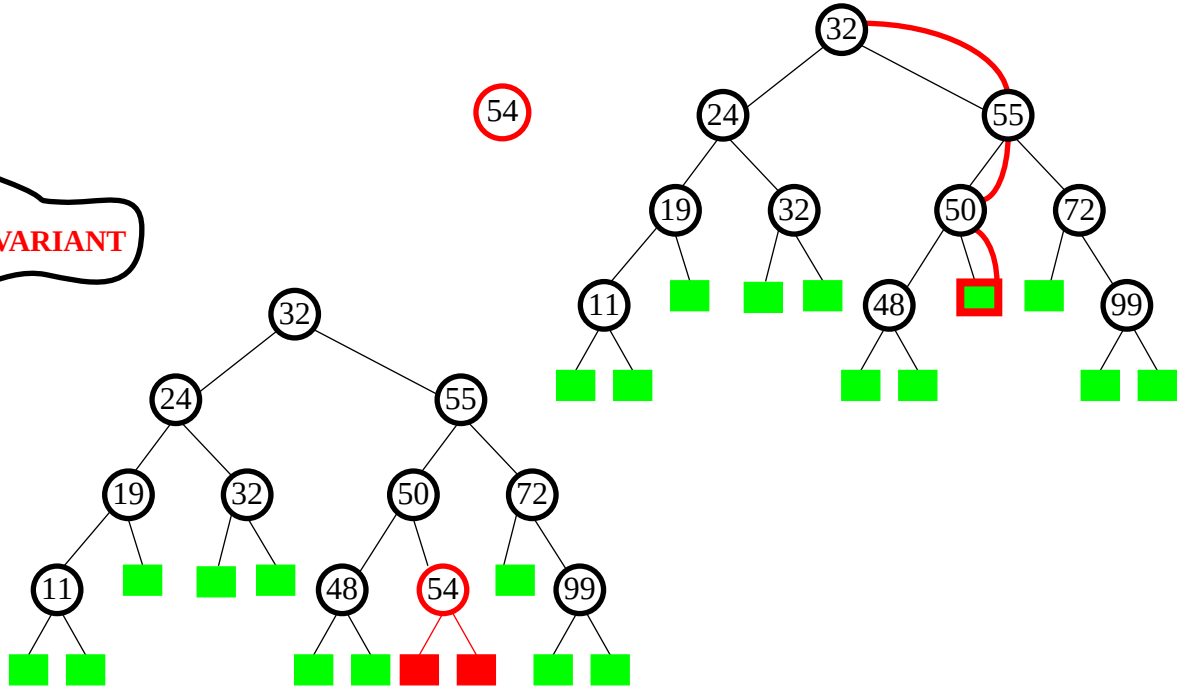
p.setElement(new Item(k,e))

insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

sikrer
BST INVARIANT



if (BST.**isExternal**(**p**))

ny Intern node: expandExternal(**p**)

p.setElement(new Item(k,e))

insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

sikrer
BST INVARIANT

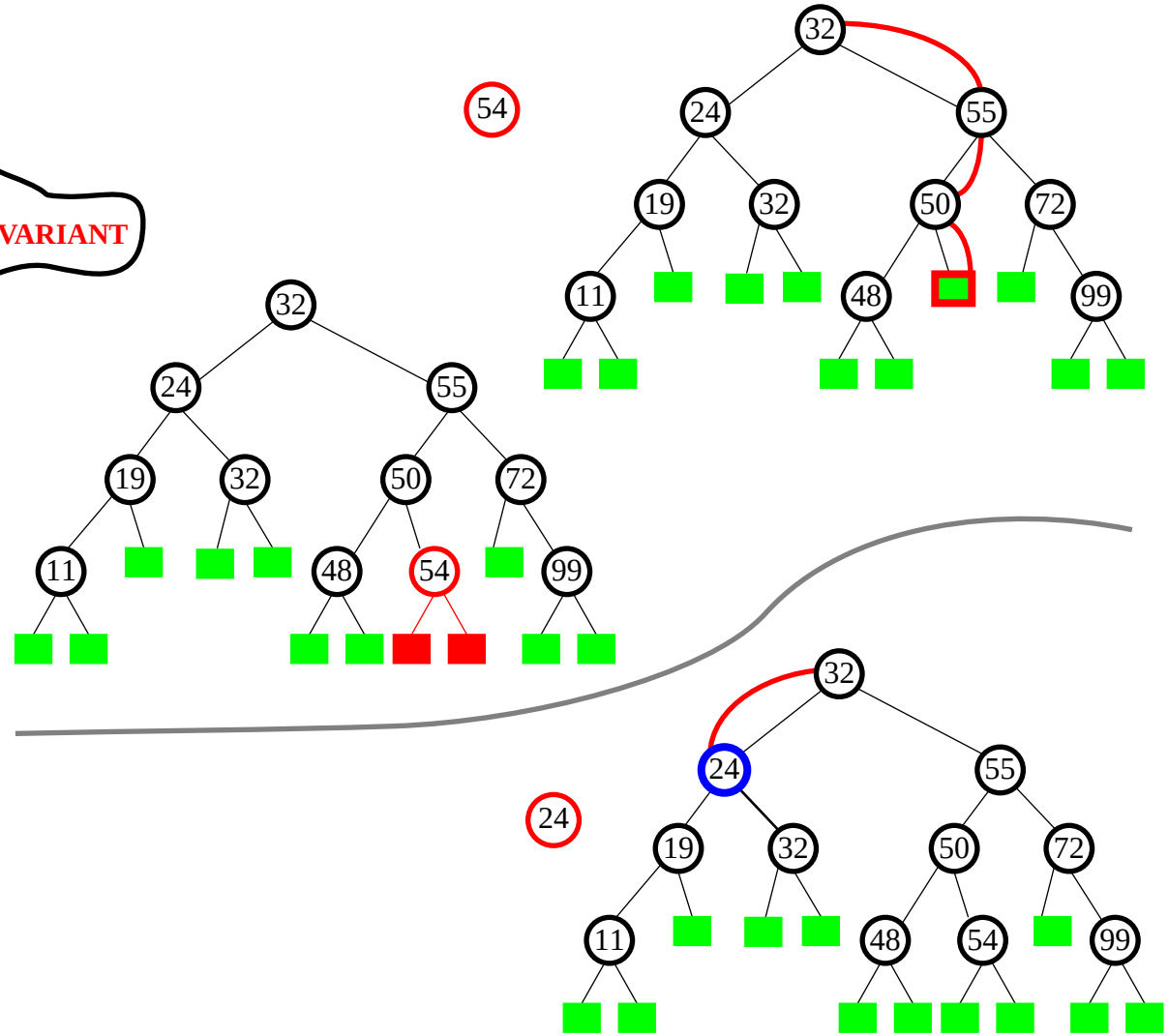
if (BST.**isExternal**(**p**))

ny Intern node: expandExternal(**p**)

p.setElement(new Item(k,e))

else

}



insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

sikrer
BST INVARIANT

if (BST.**isExternal**(**p**))

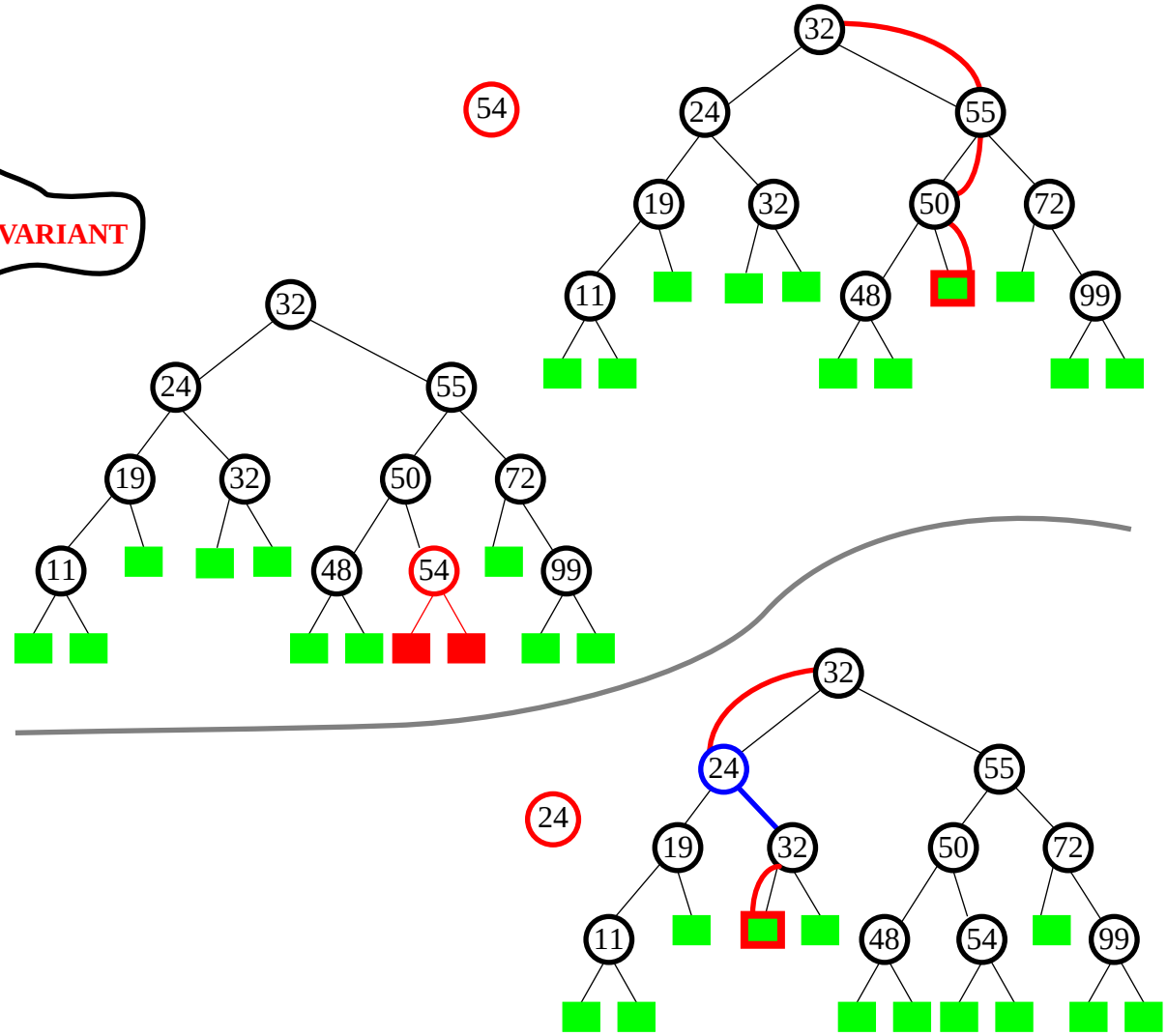
ny Intern node: expandExternal(**p**)

p.setElement(new Item(k,e))

else

settInn(BST.**rightChild**(**p**), k, e)

}

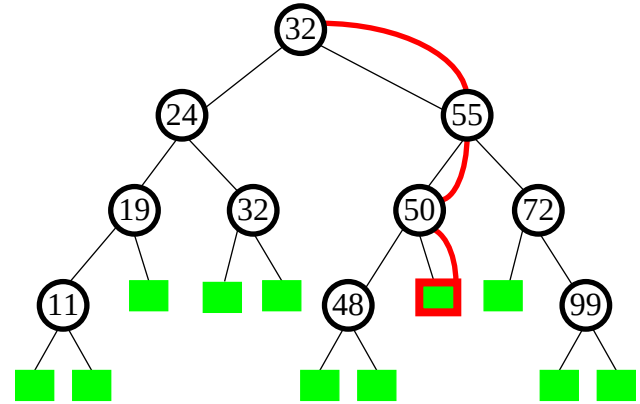


insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

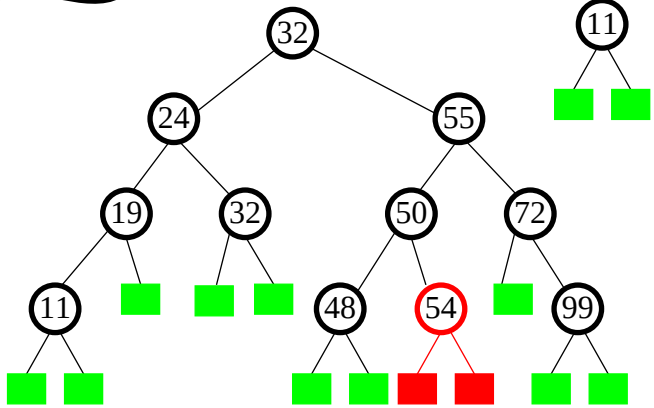
sikrer
BST INVARIANT



if (BST.**isExternal**(**p**))

ny Intern node: expandExternal(**p**)

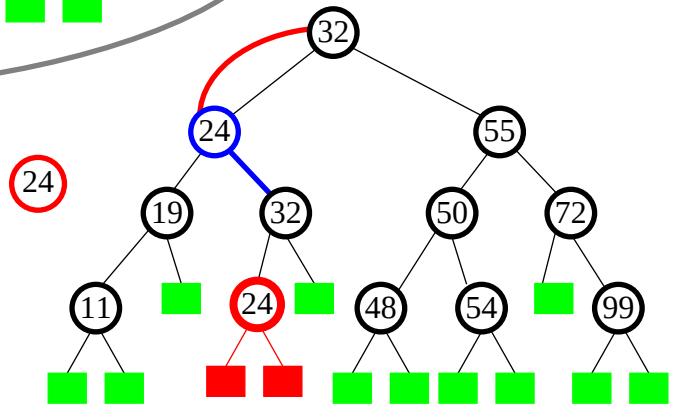
p.setElement(new Item(k,e))



else

settInn(BST.**rightChild**(**p**), k, e)

}

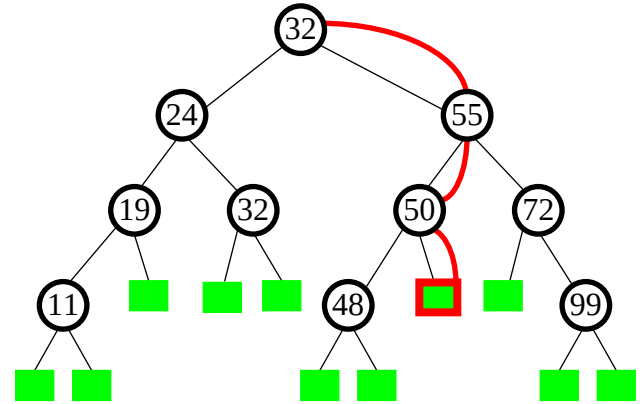


insert(Object k, o) i et BST = settInn(BST.root(), k, o)

settInn(Position v, Object k, Object o) **{**

Position **p** = **findPos**(k, v)

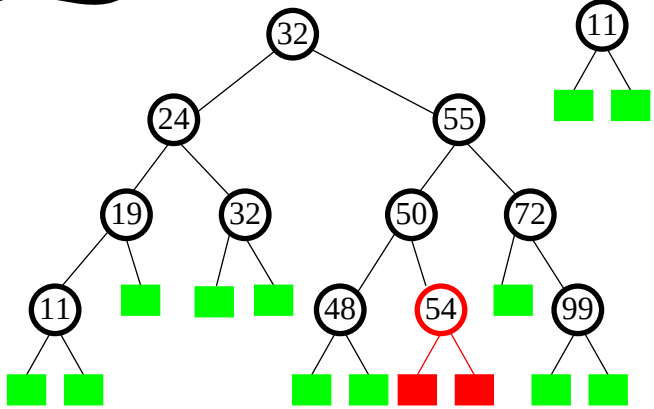
sikrer
BST INVARIANT



if (BST.**isExternal**(**p**))

ny Intern node: expandExternal(**p**)

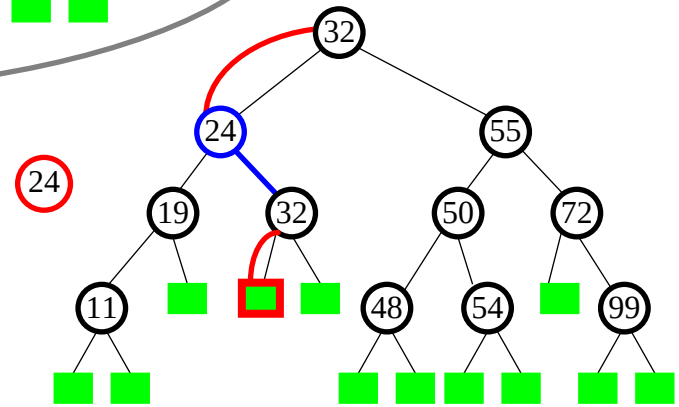
p.setElement(new Item(k,e))



else

settInn(BST.**rightChild**(**p**), k, e)

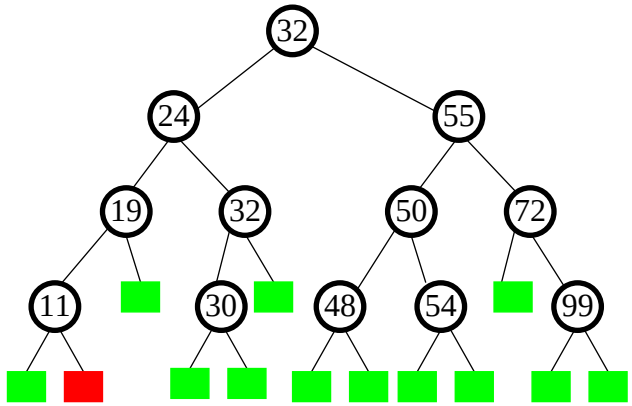
}



= O(**findPos**) = O(**h(BST)**)

Object remove(k) fra et BST

Position p = **findPos**(k)
1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey



Object remove(k) fra et BST

Position p = **findPos**(k)

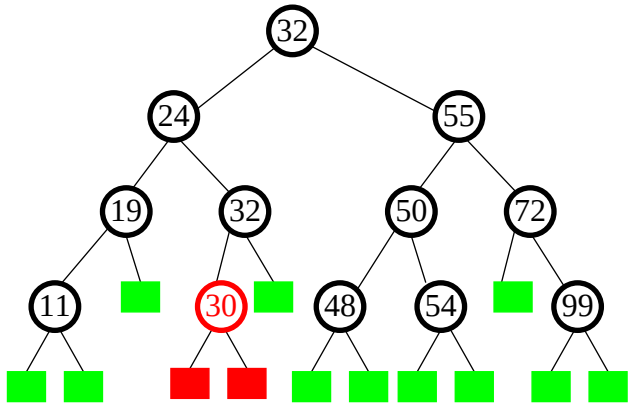
1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

return ((Item)removeAboveExt(leftChild(p)).element()



Object remove(k) fra et BST

Position p = **findPos**(k)

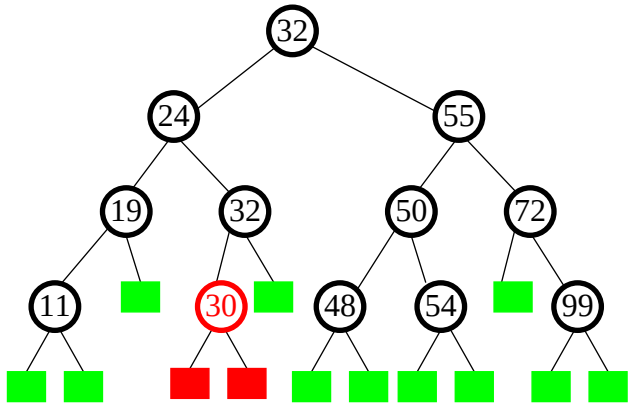
1. **if** (p er et **blad**) // isExternal(p)

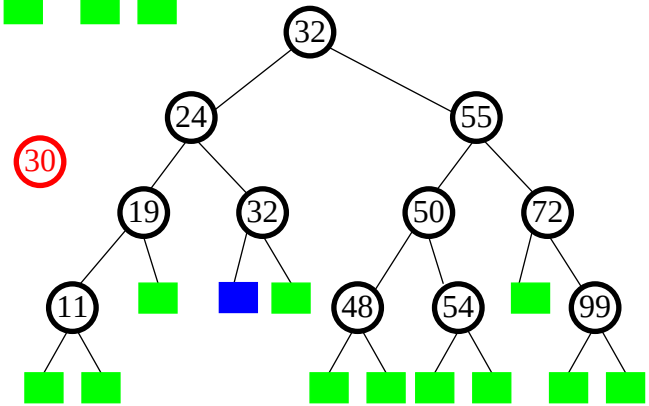
   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

return ((Item)removeAboveExt(leftChild(p)).element()





Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

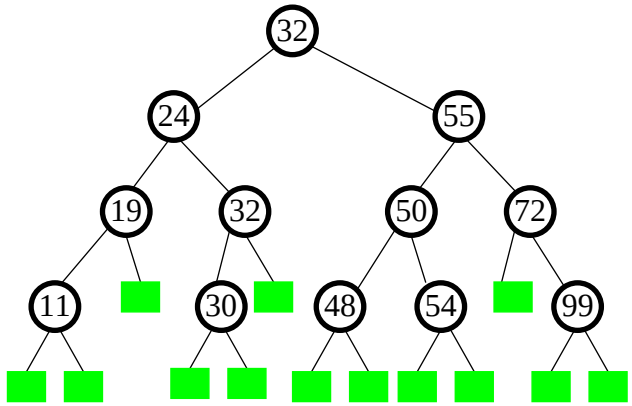
   - NoSuchKey

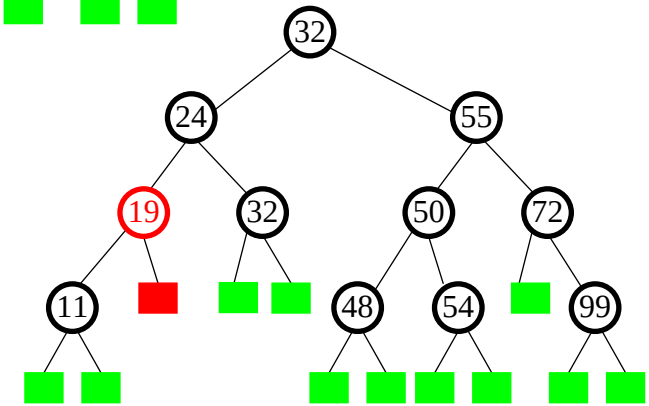
2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

return ((Item)removeAboveExt(leftChild(p)).element()

3. **else if** **nøyaktig et barn** er et **blad b**





Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

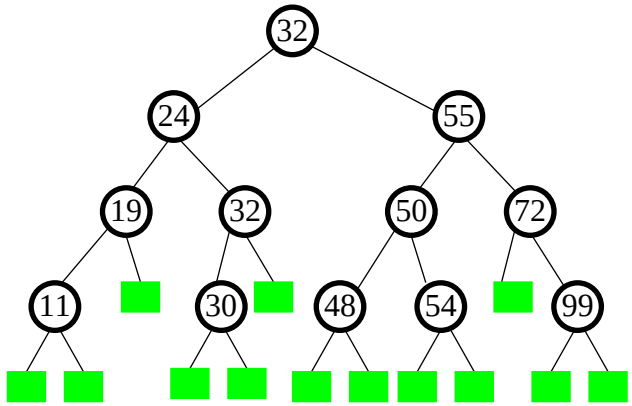
   - fjern p

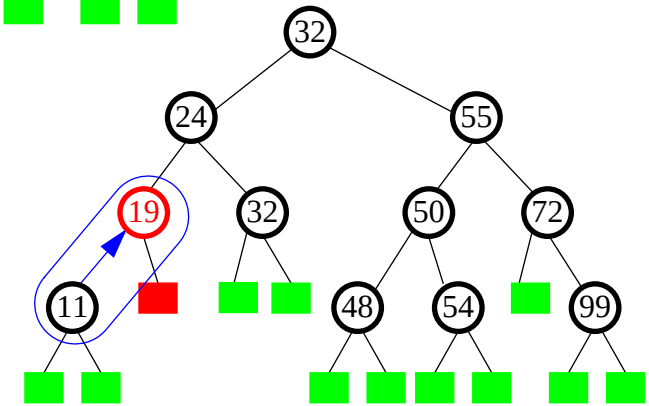
return ((Item)removeAboveExt(leftChild(p)).element()

3. **else if** **nøyaktig et barn** er et **blad b**

   - erstatt p med andre barnet

return ...removeAboveExt(**b**)...





Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

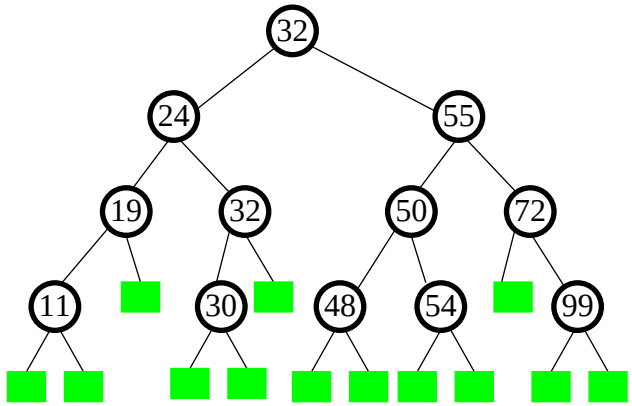
   - fjern p

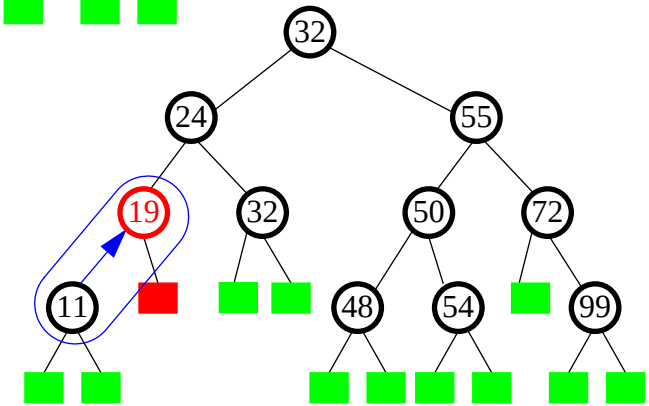
return ((Item)removeAboveExt(leftChild(p)).element()

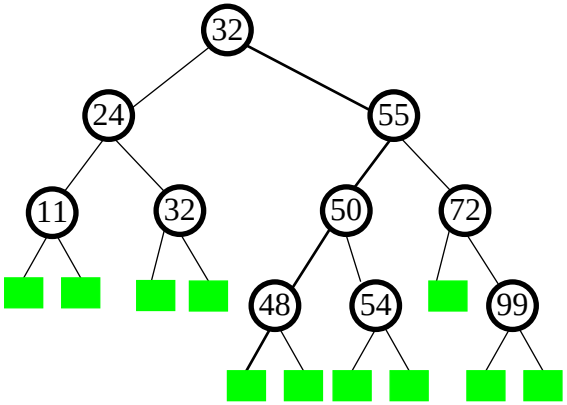
3. **else if** **nøyaktig et barn** er et **blad b**

   - erstatt p med andre barnet

return ...removeAboveExt(**b**)...







Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

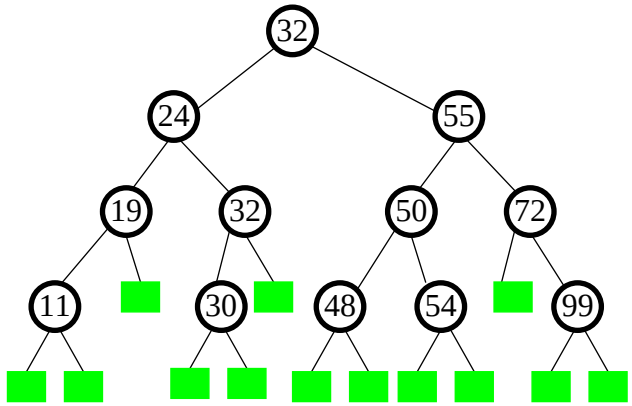
return ((Item)removeAboveExt(leftChild(p)).element()

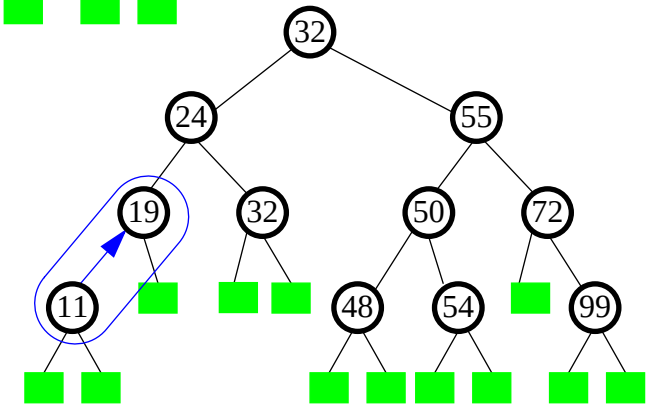
3. **else if** **nøyaktig et barn** er et **blad b**

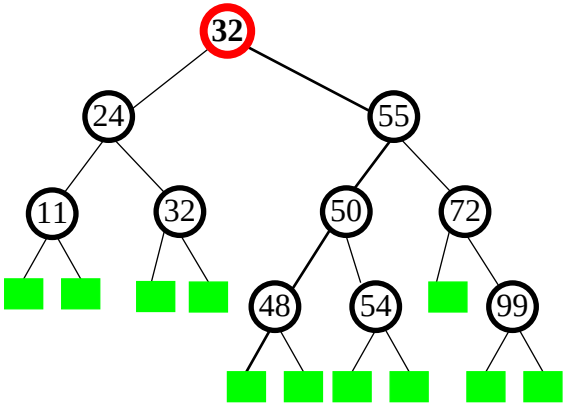
   - erstatt p med andre barnet

return ...removeAboveExt(**b**)...

4. **else** // **ingen av barna** er et **blad**







Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

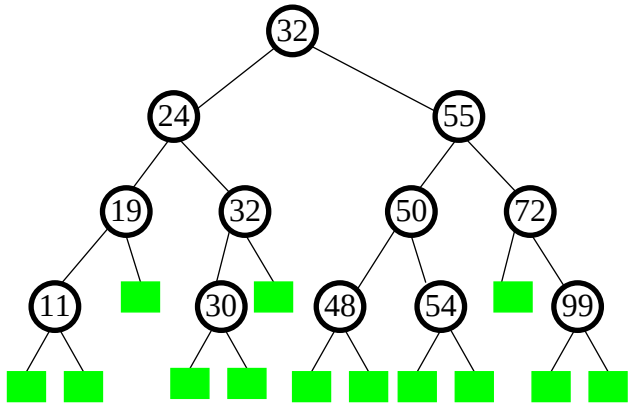
return ((Item)removeAboveExt(leftChild(p)).element()

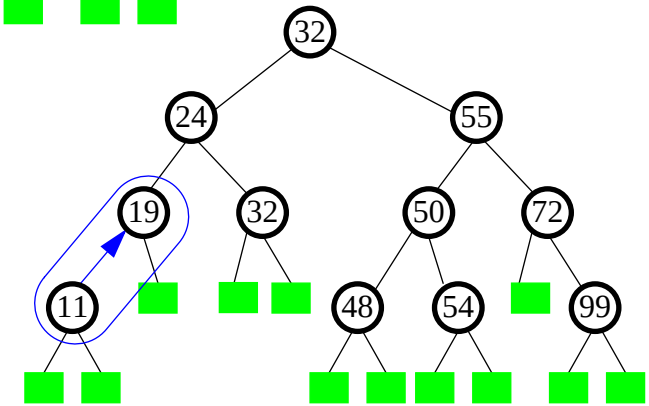
3. **else if** **nøyaktig et barn** er et **blad b**

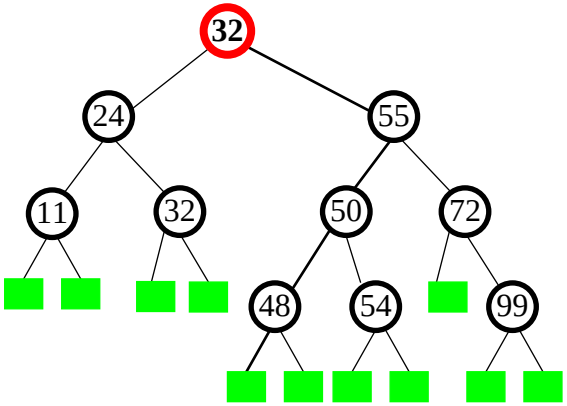
   - erstatt p med andre barnet

return ...removeAboveExt(**b**)...

4. **else** // **ingen av barna** er et **blad**







11	24	32	**32**	48	50	54	55	72	99

Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

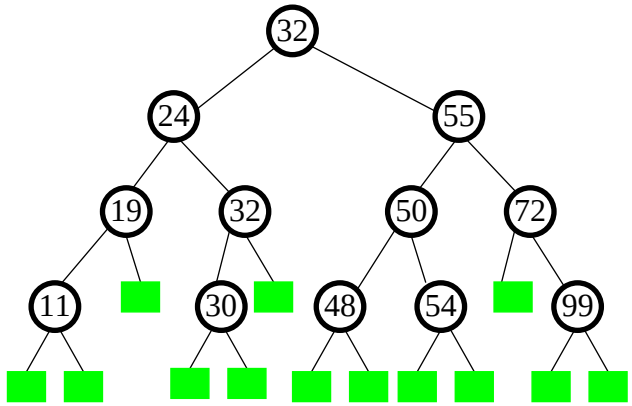
return ((Item)removeAboveExt(leftChild(p)).element()

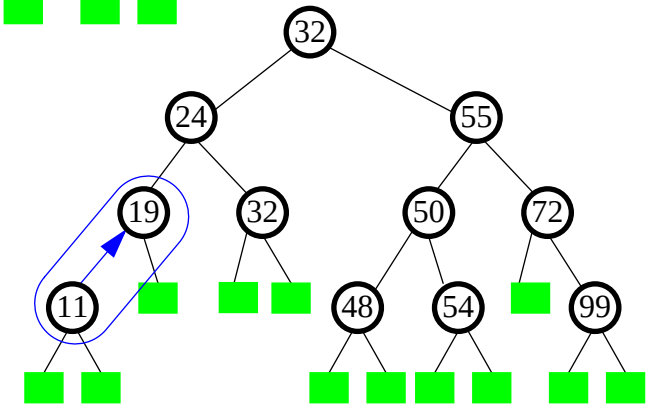
3. **else if** **nøyaktig et barn** er et **blad b**

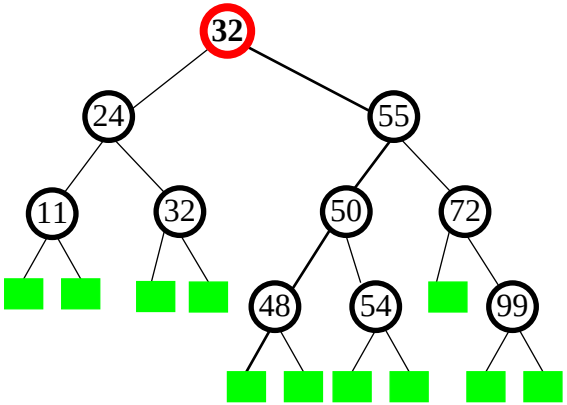
   - erstatt p med andre barnet

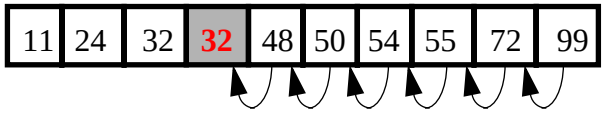
return ...removeAboveExt(**b**)...

4. **else** // **ingen av barna** er et **blad**









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

return ((Item)removeAboveExt(leftChild(p)).element()

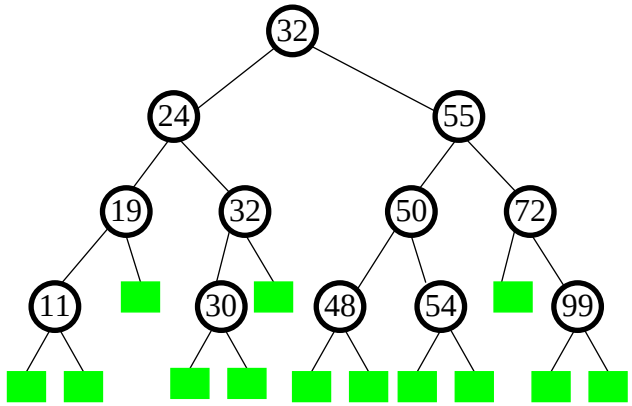
3. **else if** **nøyaktig et barn** er et **blad b**

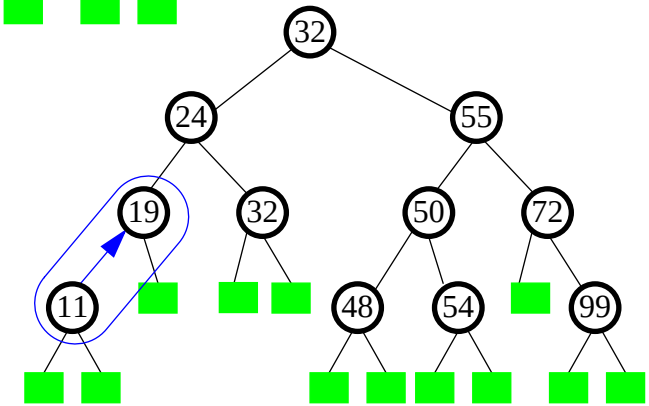
   - erstatt p med andre barnet

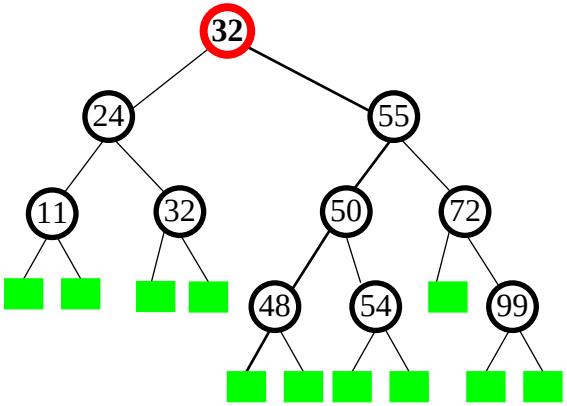
return ...removeAboveExt(**b**)...

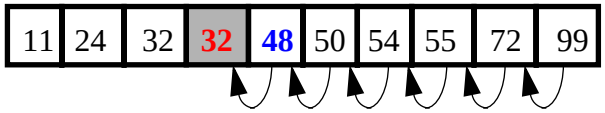
4. **else** // **ingen av barna** er et **blad**

   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)

   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**

   - fjern p

return ((Item)removeAboveExt(leftChild(p)).element()

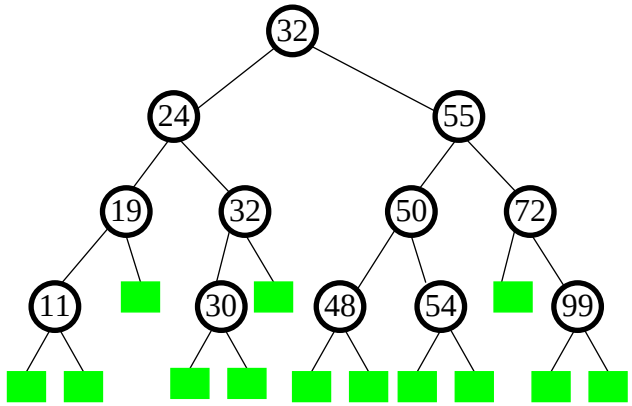
3. **else if** **nøyaktig et barn** er et **blad b**

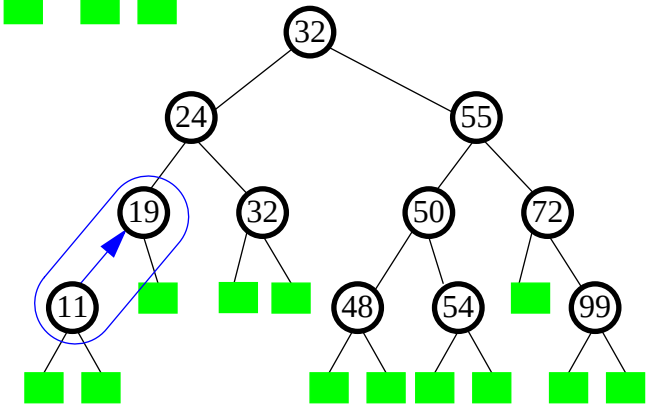
   - erstatt p med andre barnet

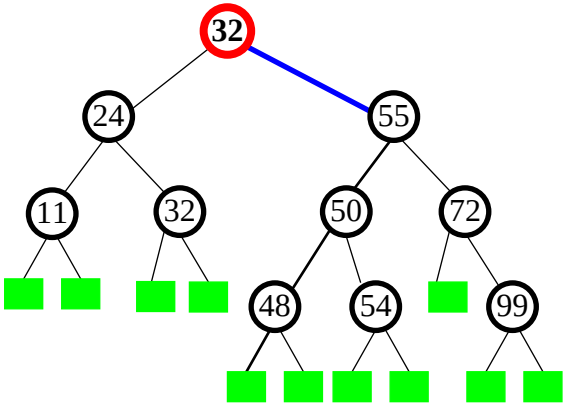
return ...removeAboveExt(**b**)...

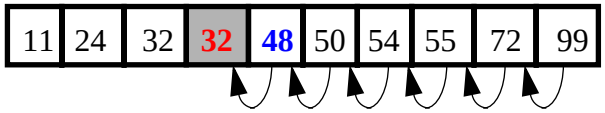
4. **else** // **ingen av barna** er et **blad**

   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p
     n = **rightChild**(p)









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

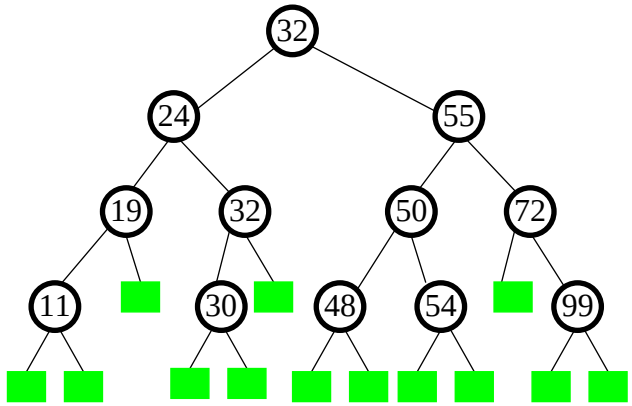
return ((Item)removeAboveExt(leftChild(p)).element()

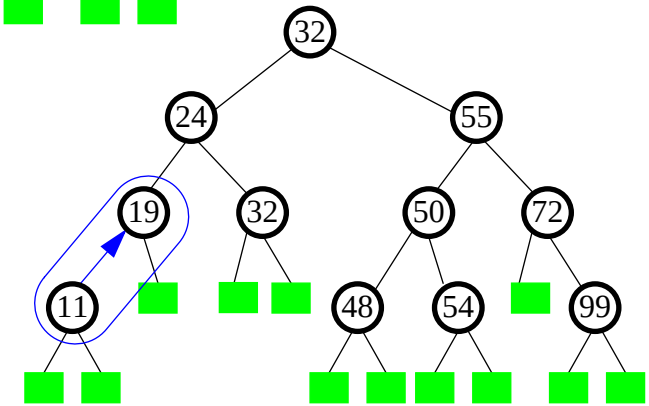
3. **else if** **nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

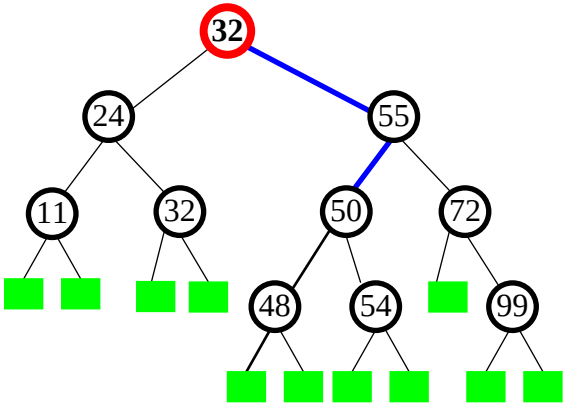
return ...removeAboveExt(**b**)...

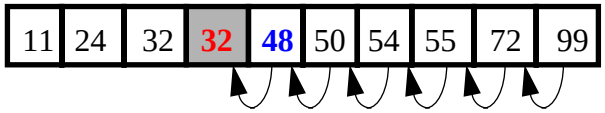
4. **else** // **ingen av barna** er et **blad**
   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p

```
n = rightChild(p)
while (! isExternal(leftChild(n)))
    n = leftChild(n)
```









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

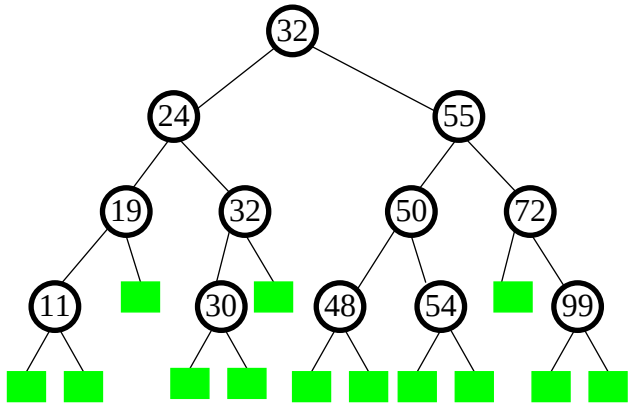
return ((Item)removeAboveExt(leftChild(p)).element()

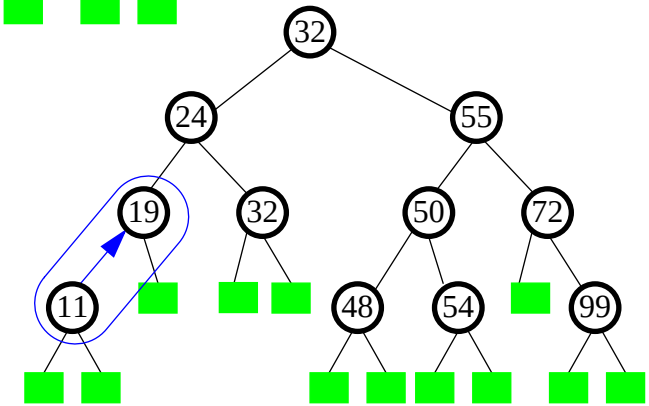
3. **else if** **nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

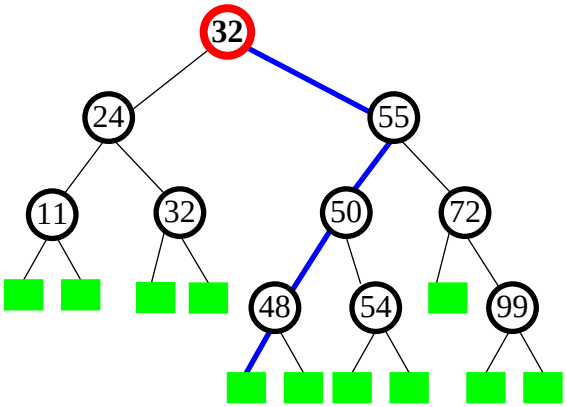
return ...removeAboveExt(**b**)...

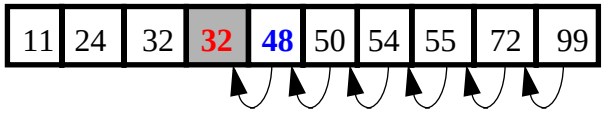
4. **else** // **ingen av barna** er et **blad**
   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p

```
n = rightChild(p)
while (! isExternal(leftChild(n)))
    n = leftChild(n)
```









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey

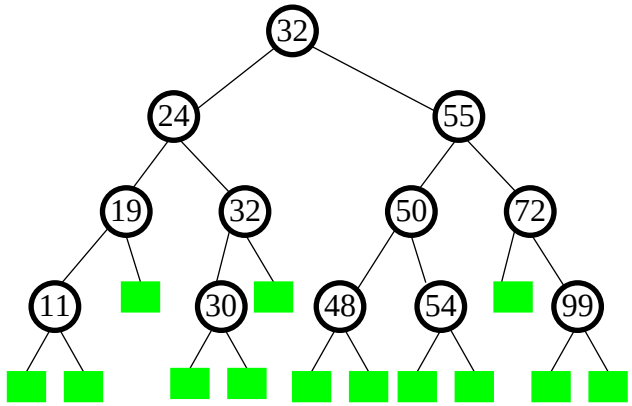
2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

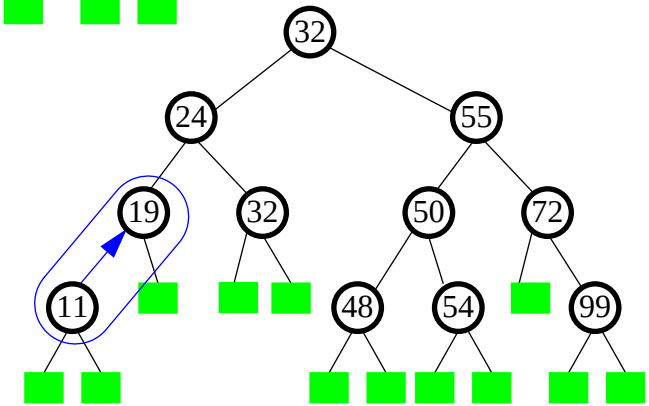
return ((Item)removeAboveExt(leftChild(p)).element()

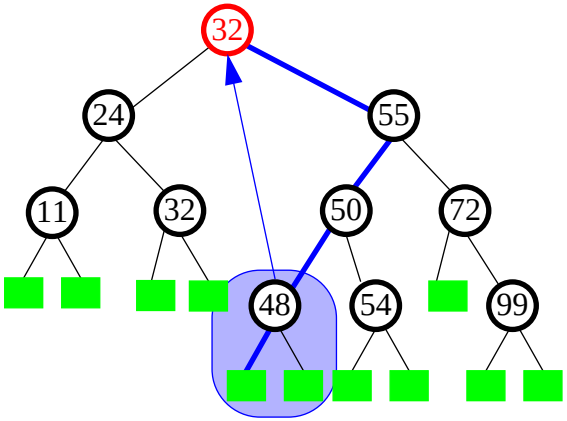
3. **else if** **nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

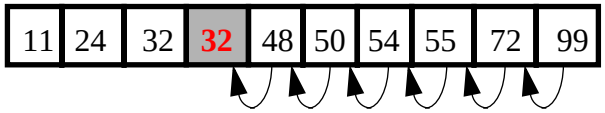
return ...removeAboveExt(**b**)...

4. **else** // **ingen av barna** er et **blad**
   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p
     n = **rightChild**(p)
     while (! isExternal(leftChild(n)))
       n = **leftChild**(n)
   - replace(p, n.element())









Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey

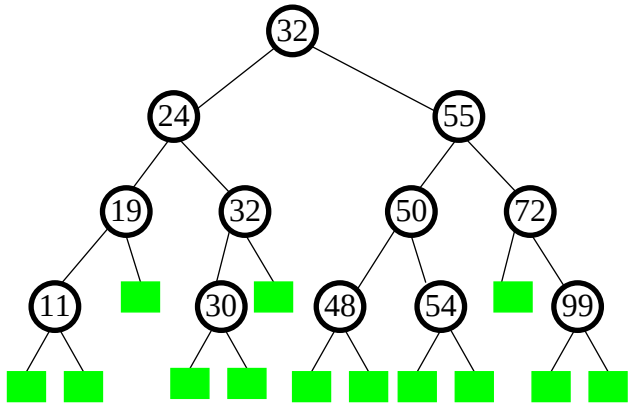
2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

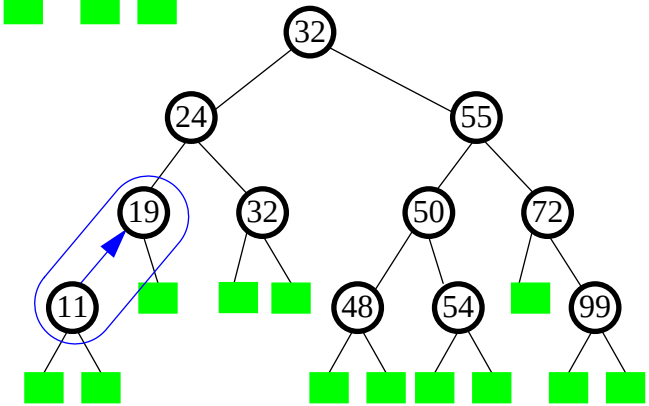
return ((Item)removeAboveExt(leftChild(p)).element()

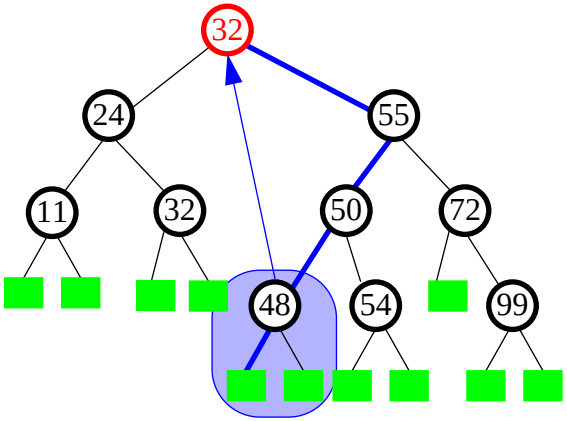
3. **else if** **nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

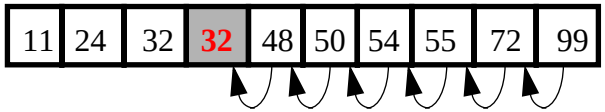
return ...removeAboveExt(**b**)...

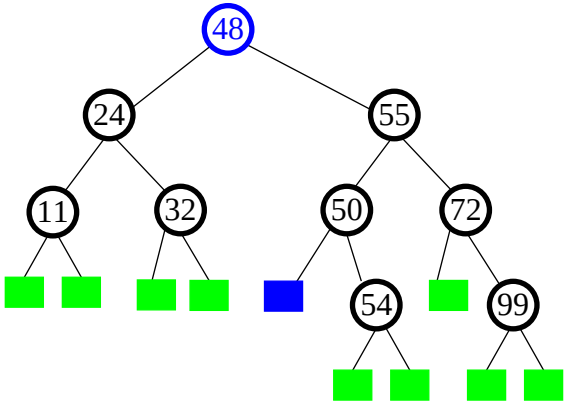
4. **else** // **ingen av barna** er et **blad**
   - hold ((Item)p.element()).element() til return
   - finn n = ‘‘neste til høyre’’ for p
     n = **rightChild**(p)
     while (! isExternal(leftChild(n)))
       n = **leftChild**(n)
   - replace(p, n.element())
   - fjern n (2. eller 3.)











Object remove(k) fra et BST

Position p = **findPos**(k)

1. **if** (p er et **blad**) // isExternal(p)
   - NoSuchKey

2. **else if** **leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

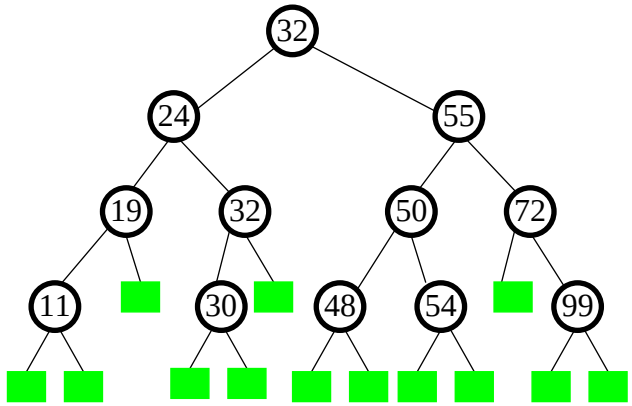
return ((Item)removeAboveExt(leftChild(p)).element()

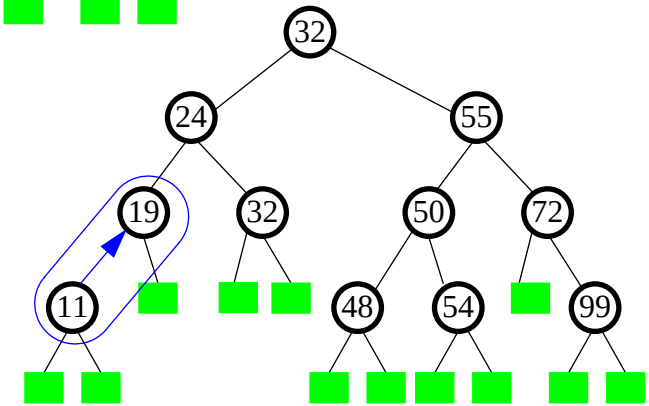
3. **else if** **nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

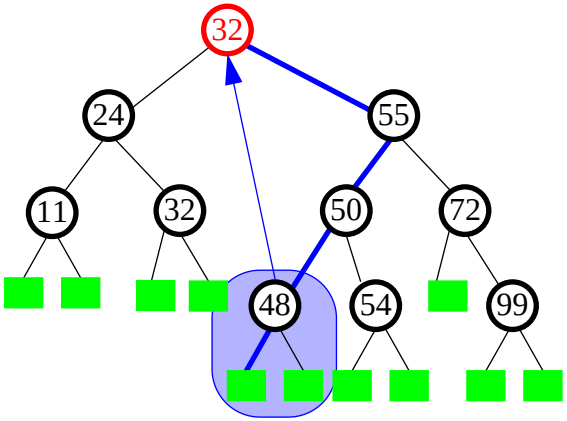
return ...removeAboveExt(**b**)...

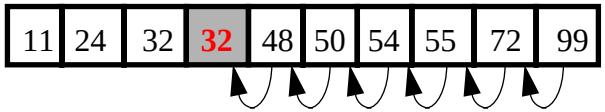
4. **else** // **ingen av barna** er et **blad**
   - hold ((Item)p.element()).element() til return
   - finn n = "neste til høyre" for p
     n = **rightChild**(p)
     while (! isExternal(leftChild(n)))
       n = **leftChild**(n)
   - replace(p, n.element())
   - fjern n (2. eller 3.)

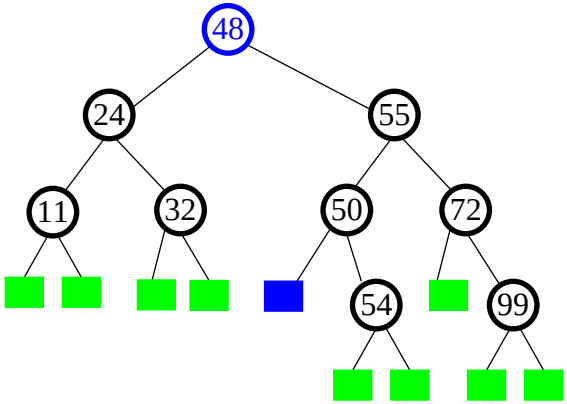
= O(**findPos**) = O(**h(BST)**)











Implementasjon av Dictionary

	usortert **Sequence**		sortert **Sequence**		BST
operasjon	**Array**	**DL**	**Array**	**DL**	
Container					
size	1	1	1	1	1
isEmpty	1	1	1	1	1
elements	n	n	n	n	n
Dictionary					
find	n	n	log n	n	hgh(n)
findAll	n	n	log n (+ s)	n	hgh(n)
insert	1	1	n	n	hgh(n)
remove	n	n	n	n	hgh(n)
closestAfter/Before	n	n	n	n	hgh(n)

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

Balansering

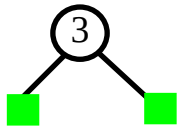
– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```

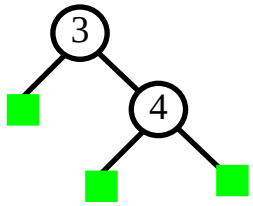


insert(**3**,A);

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```

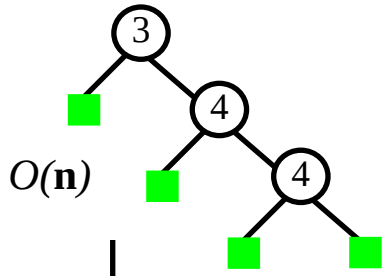


insert(**3**,A);
insert(**4**,B);

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



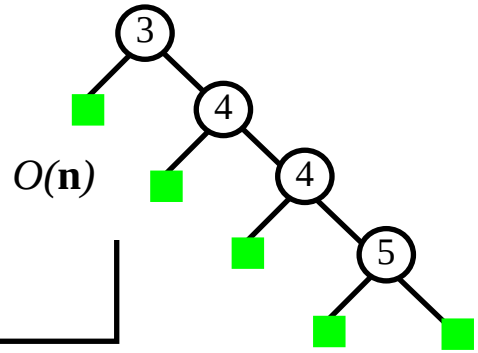
O(**n**)

insert(**3**,A);
insert(**4**,B);
insert(**4**,B);

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```

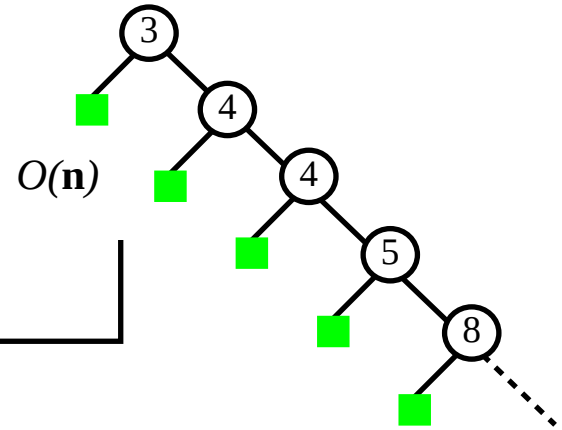


insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D);

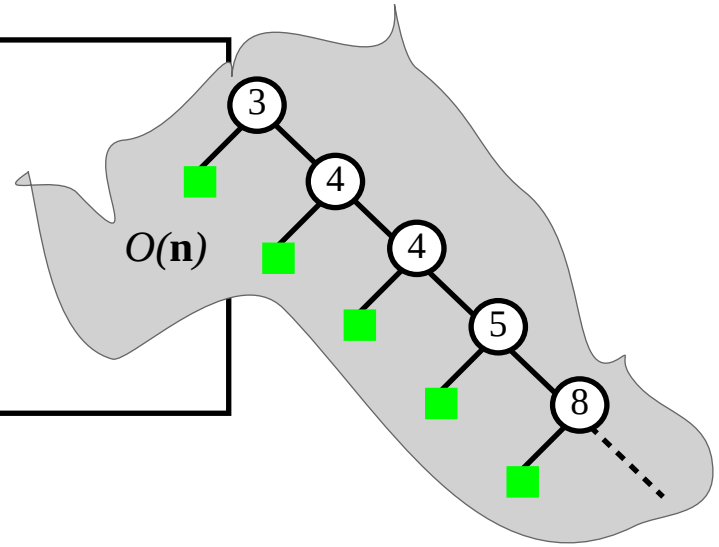
hgh(BST) = *O*(**n**)

tilsvarende uhell kan skje ved en serie remove(k)

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D);

hgh(BST) = O(**n**)

tilsvarende uhell kan skje ved en serie remove(k)

AVL Tre

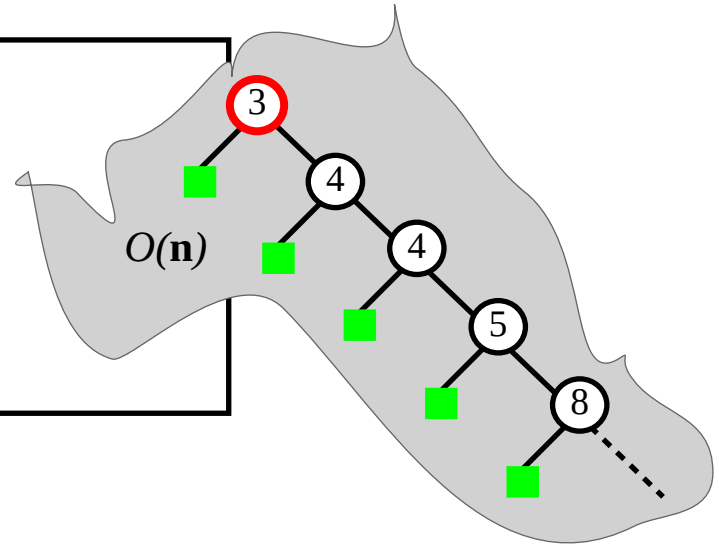
er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

1. **BST INVARIANT** (“relasjonell”) – for hver intern node *p* :
   for hver node *v* i *p*’s venstre subtre : key(*v*) ≤ key(*p*)
   for hver node *h* i *p*’s høyre subtre : key(*h*) ≥ key(*p*)

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D);

hgh(BST) = O(**n**)

tilsvarende uhell kan skje ved en serie remove(k)

AVL Tre

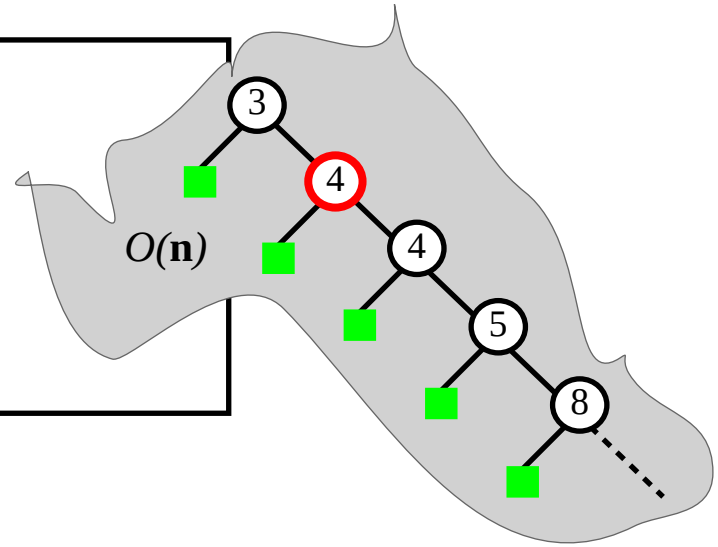
er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

1. **BST INVARIANT** (“relasjonell”) – for hver intern node *p* :
   for hver node *v* i *p*’s venstre subtre : key(*v*) ≤ key(*p*)
   for hver node *h* i *p*’s høyre subtre : key(*h*) ≥ key(*p*)
2. **AVL INVARIANT** (“strukturell”) – hver intern node *p* er **balansert** :
   | **hgh**(***leftChild***(*p*)) – **hgh**(***rightChild***(*p*)) | ≤ 1

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D);

hgh(BST) = O(**n**)

tilsvarende uhell kan skje ved en serie remove(k)

AVL Tre

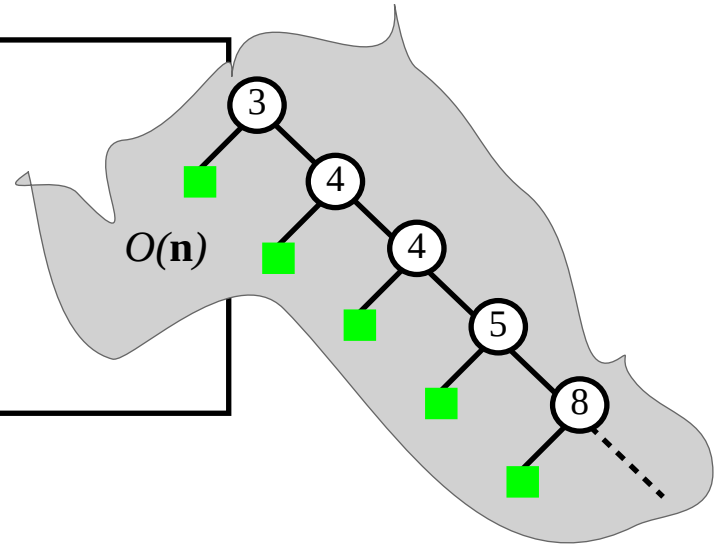
er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

1. **BST INVARIANT** (“relasjonell”) – for hver intern node *p* :
   for hver node *v* i *p*’s venstre subtre : key(*v*) ≤ key(*p*)
   for hver node *h* i *p*’s høyre subtre : key(*h*) ≥ key(*p*)
2. **AVL INVARIANT** (“strukturell”) – hver intern node *p* er **balansert** :
   | **hgh**(***leftChild***(*p*)) – **hgh**(***rightChild***(*p*)) | ≤ 1

Balansering

– *hvis BST er balansert får vi* hgh(n) = **O**(log n)
– *dette er det som skjer i gjennomsnittstilfelle (gjennomsnittskompleksistet for BST er* **O**(log n)*)*

```
settInn(Position v, Object k, Object o) {
    Position p= findPos(k,v);
    if (isExternal(p))
        expandExternal(p)
        p.setElement(new Item (k,o))
    else
        settInn(rightChild(p),k,o) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D);

hgh(BST) = O(**n**)

tilsvarende uhell kan skje ved en serie remove(k)

AVL Tre

er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler)
som tilfredstiller :

1. **BST INVARIANT** (“relasjonell”) – for hver intern node *p* :
   for hver node *v* i *p*’s venstre subtre : key(*v*) ≤ key(*p*)
   for hver node *h* i *p*’s høyre subtre : key(*h*) ≥ key(*p*)
2. **AVL INVARIANT** (“strukturell”) – hver intern node *p* er **balansert** :
   | **hgh**(***leftChild***(*p*)) – **hgh**(***rightChild***(*p*)) | ≤ 1

9.2. Et AVL Tre *T* som lagrer *n* nøkler har høyden $h(T) = O(\mathbf{log\ n})$

Oppsummering

- *Ordbok (symboltabell)*

  avbildning fra nøkler til dataobjekter

Oppsummering

- *Ordbok (symboltabell)*
  avbildning fra nøkler til dataobjekter

- *Ordbok og Ordnet Ordbok ADT*
  – *med nøkkel-data objekter*
  – *med Locator*

Oppsummering

- *Ordbok (symboltabell)*

  avbildning fra nøkler til dataobjekter

- *Ordbok og Ordnet Ordbok ADT*
  - *med nøkkel-data objekter*
  - *med Locator*

- *Implementasjon*

  Sekvens
  - *usortert (DL, Array)*
  - *sortert (DL, Array)*

Oppsummering

- *Ordbok (symboltabell)*

  avbildning fra nøkler til dataobjekter

- *Ordbok og Ordnet Ordbok ADT*

  – med nøkkel-data objekter

  – med Locator

- *Implementasjon*

  Sekvens

  – usortert (DL, Array)

  – sortert (DL, Array)

  Hash Tabeller

  – valg av hash-funksjon

  – kollisjoner

Oppsummering

- *Ordbok (symboltabell)*

  avbildning fra nøkler til dataobjekter

- *Ordbok og Ordnet Ordbok ADT*

  – med nøkkel-data objekter

  – med Locator

- *Implementasjon*

  Sekvens

  – usortert (DL, Array)

  – sortert (DL, Array)

  Hash Tabeller

  – valg av hash-funksjon

  – kollisjoner

  Binære Søketrær

  – innsetting/fjerning

  – balansering (AVL trær)