Ordbøker

I. ORDBOK / PRIORITETSKØ = SEKVENS / KØ

vilkårlig / minste nøkkel

vilkårlig / første Posisjon

II. IMPLEMENTASJON MED SEQUENCE

III. IMPLEMENTASJON MED BST (BINARY SEARCH TREE)

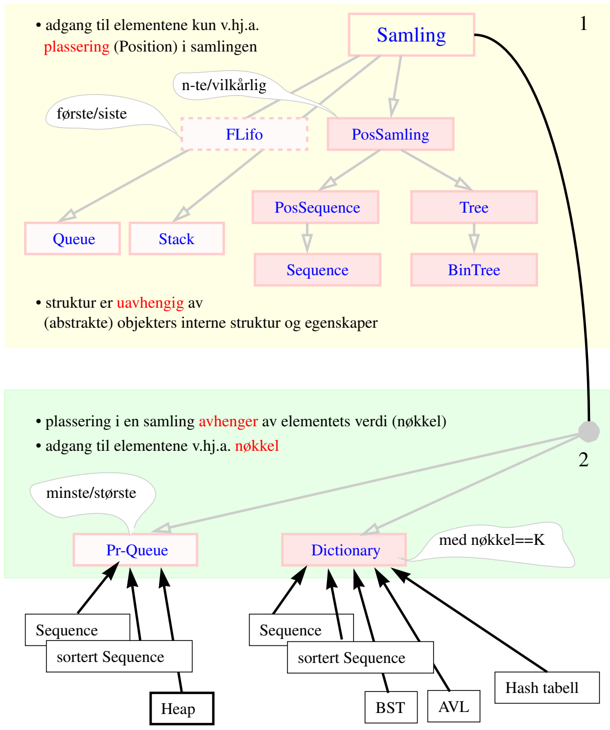
IV. IMPLEMENTASJON MED AVL-TRÆR

V. EN DIGRESJON ...

VI. IMPLEMENTASJON MED HASH TABELL

Kap. 7 (kursorisk 7.6.3; unntatt 7.5, 7.6.2, 7.7)

Ordbøker



Ordbok

en samling som gir oss **adgang til element med en vilkårlig nøkkel**

```
public interface Samling
{   int size();
    boolean isEmpty();
    Enumeration elements();
}
```

```
public interface Dictionary extends Samling {
/**  sett inn et nytt element
    * @param e Objektet som skal innsettes
    * @param k noekkel Objekt */
    void insert(Object k, Object e);
/**  finn elemenetet med en gitt nøkkel
    * @param k noekkelen til Objektet som skal finnes */
    * @return elementet med noekkelen k */
    Object find(Object k);
/**  finn alle elementer med en gitt nøkkel
    * @param k noekkelen til Objektet som skal finnes */
    * @return alle elementer med noekkelen k */
    Enumeration findAll(Object k);
/**  fjern - og returner - elementet med en gitt nøkkel
    * @param k noekkelen til Objektet som skal fjernes */
    * @return elementet med noekkelen k */
    Object remove(Object k);
/**  fjern, og returner, alle elementer med en gitt nøkkel
    * @param k noekkelen til elementer som skal fjernes */
    * @return alle elementer med noekkelen k */
    Enumeration removeAll(Object k);
}
```


Ordnet Ordbok

En Dictionary trenger kun likhetssammenlikning på nøkler

men ofte er nøkler totalt ordnet –

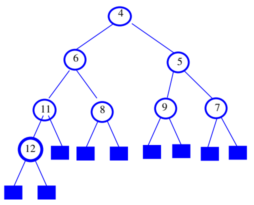
```
public interface Comparator {
    boolean lt(Object a, Object b);
    boolean leq(Object a, Object b);
    boolean gt(Object a, Object b);
    boolean geq(Object a, Object b);
    boolean eq(Object a, Object b);
    ... }
```

Dictionary vil da inneholde også metoder for 'ordnet traversering' :

```
public interface OrdDictionary extends Dictionary                {
/**  returner nøkkel før k
    * @param k noekkel Objekt */
    * @return nøkkel Objektet 'like før' k iflg. Comp */
    Object keyBefore(Object k);
/**  returner elementet med nøkkel før k
    * @param k noekkel Objekt */
    * @return elementet med nøkkel keyBefore(k) */
    Object elemBefore(Object k);
/**  tilsv. keyBefore(k) */
    Object keyAfter(Object k);
/**  tilsv. elemBefore(k) */
    Object elemAfter(Object k);
}
```

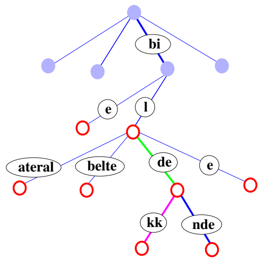
Implementasjon

ikke Haug



hvordan finner man ‘11’ ... ?

ikke Tries

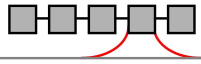


hva om nøkler ikke er String ... ?

II. Dictionary >> Sequence >> (LL,DL)

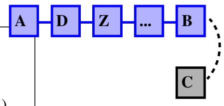
```
class DictSeq implements PrQueue {
  protected Comparator cp;
  protected Sequence sq=new DLSeq();
  public DictSeq(Comparator c)
      { cp= c; }
... }
```

!! nesten lik men dog ulik Comp i PrQueue



```
public class Par {
  private Object el, key;
  protected Par(Object k, Object e) {
     setElem(e); setKey(k); }
  public Object key() { return key;}
  public Object elem() { return el;}
  public void setKey(Object k) {key=k;}
  public void setElem(Object e) {el=e;}
```

I. DATA INVARIANT : INGEN – USORTERT



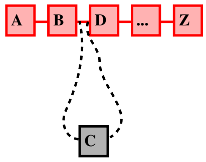
finn(k) :

```
Position c= sq.first();
boolean fant= false;
while ( c != null && ! fant )
   if (cp.eq(k, ((Par)c.elem()).key() )
       fant= true;
   else c= sq.after(c)
if (!fant) throw NoKeyException(“...”)
return c ...
```

.... **O(n)**

rem(k) :	finn og fjern ...	**O(n)**
findAll(k) :	finn ...	**Θ(n)**
remAll(k) :	finn og fjern ...	**Θ(n)**
insert(k,o) :	insertLast(new Par(k,o))	**O(1)**

II. DATA INVARIANT : SORTERT



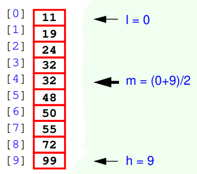
find(k) :	finn ...	**O(n)**
rem(k) :	finn og fjern ...	**O(n)**
findAll(k) :	finn ...	**O(n)**
remAll(k) :	finn og fjern...	**O(n)**
insert(k,o) :	finn og sett inn ...	**O(n)**

Dictionary >> Sequence >> Array

III. DATA INVARIANT :
Sortert Array A[l...h] :

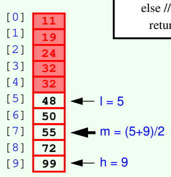
- **hvis** i < j **så cp.leq**(A[i], A[j])
- **hvis cp.lt**(A[i], A[j]) **så** i < j
  hvis cp.gt(A[i], A[j]) **så** i > j

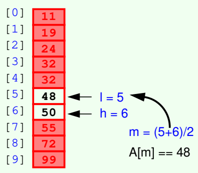
< cp.lt



BinærSøk

```
finn(k) : finn(k, l, h)
  if ( l > h ) return -1
  else
     m = (l+h) / 2
     if ( cp.eq(A[m],k) )
       return m
     else if ( cp.lt(A[m],k) )
       return finn(k,m+1,h)
     else //  cp.gt(A[m],k)
       return finn(k,l,m-1)
```



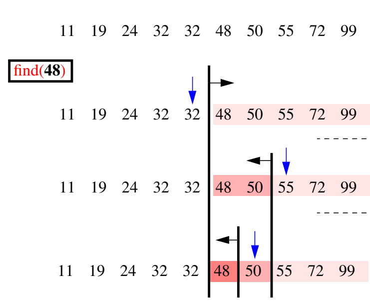


find(k) >> BinærSøk = O(log n)

Dictionary implementasjon (så langt)

operasjon	usortert Sequence Array	usortert Sequence a. DL	sortert Sequence c. Array	sortert Sequence b. DL
Samling				
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
rem	n	n	n	n
remAll	n	n	n	n

Binær Søk → Binære Søketrær

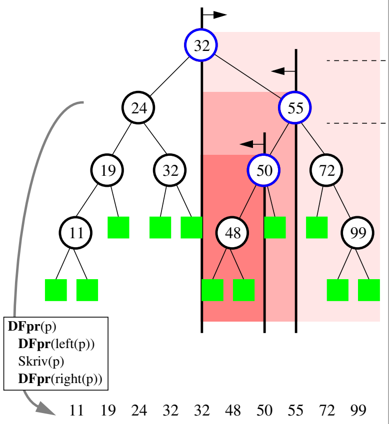


for alle **m**, *x* :
- A[*x*] > A[**m**] : *x* > **m**
- A[*x*] < A[**m**] : *x* < **m**

: A[**m**]=32 < **48**
søk til høyre

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

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



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

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

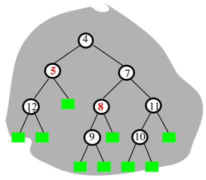
key(**m**)=50 > **48**
søk i venstre subtre

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

Binære Søketrær

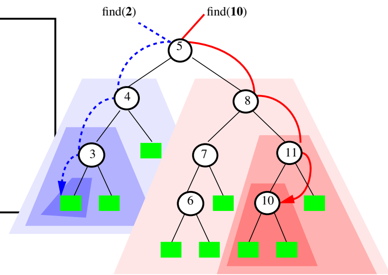
er et Binært Tre *T* (for lagring av nøkler, eller objekter med nøkler) som tilfredstiller :

BST INVARIANT ("relasjonell")
for enhver node *p* :
  for enhver node *v* i *p*'s **venstre subtre** : cp.leq(key(*v*), key(*p*)) ≤
  for enhver node *h* i *p*'s **høyre subtre** : cp.geq(key(*h*), key(*p*)) ≥
eller:
  DFS (inOrder) enumerering av noder gir en ordnet sekvens



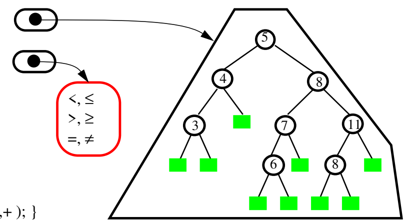

```
findP(k) : findP(k,T.root())
findP(Object k, Position p)
  if ( isExternal(p) ) - k finnes ikke
  else if ( cp.eq(k,key(p)) ) - funnet
  else if ( cp.lt(k,key(p)) )
      // let videre i venstre subtre
      findP(k,leftChild(p))
  else   // cp.gt(k,key(p))
      // let videre i høyre subtre
      findP(k,rightChild(p))
```

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



III. Implementasjon av Dictionary (BST)

```
public class BSTDict implements Dictionary {
protected  BinTree BST
protected Comparator cp
 ...
```



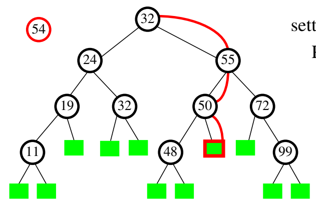

```
  protected boolean DI()
  { return Nok(BST.root(),- ,+ ); }

  private boolean Nok(Position p, Object l, Object h)
  {   if (BST.isExternal(p)) return true;
      else if ( cp.lt(key(p),  l) || cp.gt(key(p), h) ) return false;
      else return         (Nok(BST.leftChild(p), l, key(p)) &&
                           Nok(BST.rightChild(p), key(p),h)) }

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

  public Object find(Object k)
      { Position p = findP(k) ;
          if ( BST.isExternal(p) ) throw new NoKeyException("...");
          return ((Par) p.elem()) . elem() ; }

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

```
findAll(Object k, Position p, Enum e) {
   f= findP(k,p);
   if (p != null && BST.isInternal(p))
       e.add(p.elem());
       findAll(k, BST.leftChild(p), e);
       findAll(k, BST.rightChild(p), e);
}
```

```
  public Position insert(Object k, Object e)  { ... }
  public Object rem(Object k)  { ... }
  public Enumeration remAll(Object k)  { ... }
  ...
```

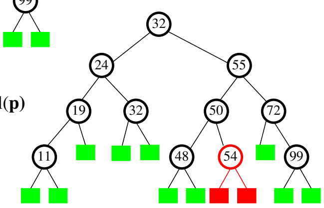

insert(Object k, e) i et BST :

settInn(BST.root(), k, e)

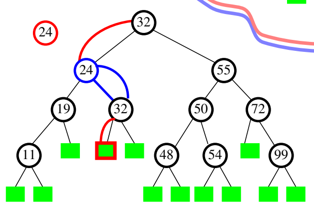


settInn(Position v, Object k, e) {
  Position **p** = **findP**(k, v)

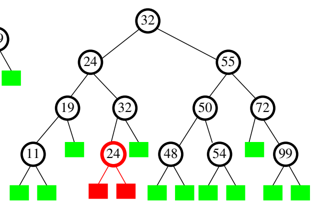
sikrer
BST INVARIANT



if (BST.**isExternal**(**p**))
  ny Intern node: expandExternal(**p**)
  p.setElem(new Par(k,e))



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

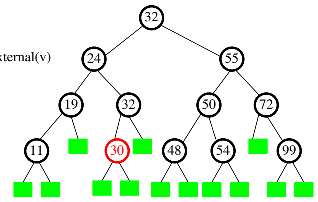


= *O*(**findP**) = *O*(**h(BST)**)

rem(k) fra et BST

Position p = **findP**(k)

1. **if** (p er et **blad**) // v er et blad = isExternal(v)
   - NoKeyException

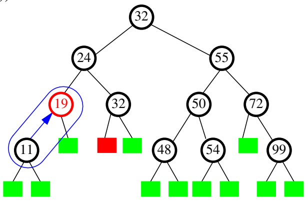


2. **else if leftChild**(p) og **rightChild**(p) er **blader**
   - fjern p

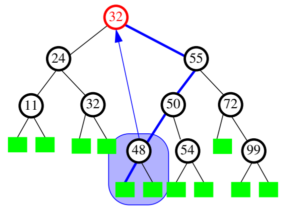
(return ((Par)removeAboveExt(leftChild(p)).elem())

3. **else if nøyaktig et barn** er et **blad b**
   - erstatt p med andre barnet

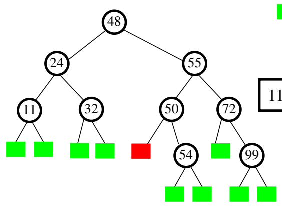
(return ...removeAboveExt(**b**)...)



4. **else // ingen av barna** er et **blad**
   - hold ((Par)p.elem()).elem() til return
   - finn n = "neste til høyre" for p
     n = **rightChild**(p)
     while (! isExternal(leftChild(n)))
     n = **leftChild**(n)
   - replace(p, n.elem())
   - fjern n (2. eller 3.)



11	24	32	**32**	48	50	54	55	72	99



= O(**findP**) = O(**h**(**BST**))

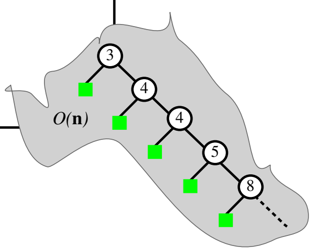
BST-Implementasjon av Dictionary

```
public class DictBST implements Dictionary {
  protected BinTree BST; protected Comparator cp;
  protected boolean DI() { return Nok(BST.root(),- ,+ ); }
  protected Position findP(Object k) {return findP(k, BST.root());}
  protected Position findP(Object k, Position p)
  {  if ( BST.isExternal(p) ) return null
     else if ( cp.eq(k, key(p)) ) return p
     else if ( cp.lt(k, key(p)) ) return findP(k, BST.leftChild(p))
     else return findP(k, BST.rightChild(p)) }
  public Object find(Object k)
  {  Position p = findP(k)
     if ( BST.isExternal(p) ) throw new NoKeyException("...")
     return ((Par) p.elem()) . elem() ; }
  public Enumeration findAll(Object k)
  {  E= new Enum(); findAll(k, BST.root(), E) ; return E; }
  public Position insert(Object k, Object e) { return settInn(BST.root(), k, e) }
  protected Position settInn(Position p, Object k, Object e)
  {  Position p= find(k)
     if (BST.isExternal(p)) { BST.expandExtern(p); p.setElem(new Par(k,e)); return p }
     else return settInn(BST.rightChild(p), k, e); }
  public Object rem(Object k)
  {  Position p= find(k)
     if (BST.isExternal(p)) throw ...
     else if (begge barna BST.isExternal(...))
        return ((Par)BST.removeAboveExt(p).elem()).elem();
     else if (et barn BST.isExternal(...)) { erstatt p med andre barnet; return ... }
     else { finn neste til høyre; erstatt .... } }
  public Enumeration remAll(Object k) { ... }
... }
```


Balansering

find(k), insert(k,e), rem(k) er alle O(**h**(**BST**)) som bør være O(**log n**)

```
settInn(Position v, Object k, Object e) {
   Position p= findP(k,v);
   if (isExternal(p))
      – en ny intern node
      – sett inn (k,e) der
   else
      – settInn(rightChild(p),k,e) }
```



insert(**3**,A);
insert(**4**,B);
insert(**4**,B);
insert(**5**,C);
insert(**8**,D); **h**(**BST**) = O(**n**)

tilsvarende uhell kan skje ved en serie rem(k)

AVL Trær

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

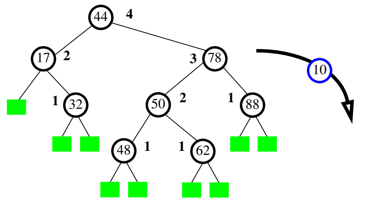
1. **BST Invariant** ("relasjonell") – for hver intern node p :
   for hver node v i p's venstre subtre : key(v) ≤ key(p)
   for hver node h i p's høyre subtre : key(h) ≥ key(p)
2. **AVL Invariant** ("strukturell") – hver intern node p er **balansert** :
   | **height**(***leftChild***(p)) – **height**(***rightChild***(p)) | ≤ 1

7.2. Et AVL Tre T som lagrer n nøkler har høyden $h(T) = O(\mathbf{log\ n})$

IV. AVL-Implementasjon av Dictionary

```
public class DictAVL extends DictBST {

//   protected BinTree BST;   nå skal dette være et AVL tre

//   protected Comparator cp;

     protected boolean DI()
        { super . DI()  &&  AVL-balansering }

//   protected Position findP(Object k)

//   protected Position findP(Object k, Position p)

//   public Object find(Object k)

//   public Enumeration findAll(Object k)

     public Position insert(Object k, Object e)
        { super . insert(k, e)  &&  AVL-balansering }

//   protected Position settInn(Position p, Object k, Object e)

     public Object rem(Object k)
        { super . remove(k)  &&  AVL-balansering }

     public Enumeration remAll(Object k) { ... }

... }
```

insert(k,e) i et AVL Tre



1. super . insert(k,e);

if (AVL invariant er ok)

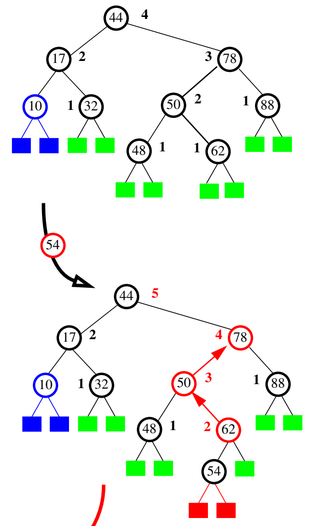
– ferdig ;

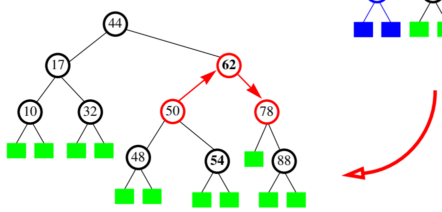
else // ubalanse !!!

2. Gjenopprett

2.a) **AVL INVARIANTEN ;**

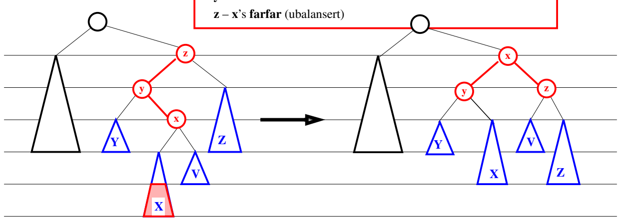
2.b) **BST INVARIANTEN ;**





2. Rotasjon(x,y,z)

p – ny insatt node
ubalanse oppstår kun på stien **S** fra **p** til roten
x – første noden på **S** (over el. lik **p**) hvis **farfar** er ubalanser
y – x's **far**
z – x's **farfar** (ubalansert)



inorder (DFS) **liste**	**a**		**b**		**c**			**(1)**
av noder:	y		x		z			
inorder (DFS) **liste**	**T1**		**T2**		**T3**		**T4**	
av subtrær rotet under x, y, z	Y		X		V		Z	
BST invariant :	**T1**	**a**	**T2**	**b**	**T3**	**c**	**T4**	
her	Y	y	X	x	V	z	Z	

z = **b** – et nytt subtre med b i roten **(2)**

b.leftChild = **a** ; **b**.rightChild = **c** ;

// **a b c**

a.leftChild = **T1**; **a**.rightChild = **T2** ;

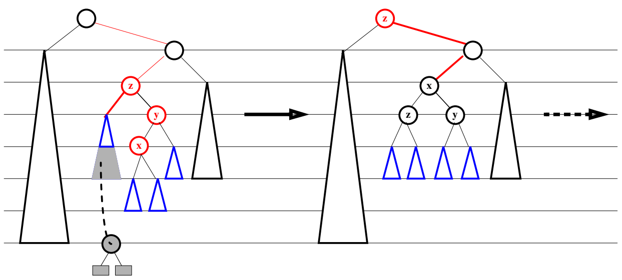
// **T1 a T2 b c**

c.leftChild = **T3** ; **c**.rightChild = **T4** ;

// **T1 a T2 b T3 c T4**

rem(k) fra et AVL Tre

1. **super . rem(k)** ; if (AVL invariant er ok) – ferdig



2. **else**:

p – faren til fjernet node
ubalanse oppstår på stien **S** fra **p** til roten
z – første noden på **S** (over **p**) som er ubalanser
y – z's **høyste barn** (ligger ikke på **S**)
x – y's **høyste barn** (ikke alltid entydig)

fortsett mot roten
finner du en ubalansert z
gjenta 2.

rotasjon(x, y, z)

$= O(\textbf{height(AVL)}) = O(\textbf{log n})$

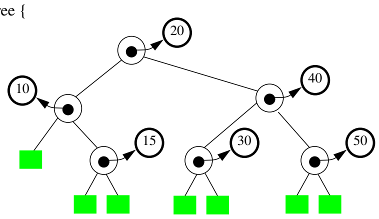
AVL–Implementasjon av Dictionary (O(log n))

```
public class AVLDict extends BSTDict {
//   protected BinTree BST;  AVL
//   protected Position findP(Object k)
//   find(Object k), findAll(Object k),...

  protected boolean DI()
     { return super.DI() && erBalansert(); }

  private boolean erBalansert()
     //  sjekk rekursivt at for hver node p erBalansert(p)

  private boolean erBalansert(Position p)
     {  pl= heigh(BST.leftChild(p))              pl= ((AVLPar) BST.leftChild(p).elem()) . height()
        pr= heigh(BST.rightChild(p))             pr= ((AVLPar) BST.rightChild(p).elem()) . height()
        return (abs(pl - pr) <= 1) }

  private void rotasjon(Position x, y, z) { ... }

  public Position insert(Object k, Object e)
     { p = super.insert(k,e);                    BST.replace(p, new AVLPar(k,e,1) )
        sjekk balanse over p                     gå oppover, oppdater - setH(..),
        - roter om nødvendig }                   finner du ubalanse - rotasjon(...)

  public Object rem(Object k)
     { o = super.rem(k);
        sjekk balanse helt til roten             gå oppover, oppdater - setH(..),
        - roter om nødvendig }                   finner du ubalanse - rotasjon(...),
                                                 fortsett helt til BST.root()
```

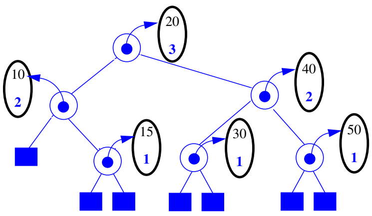
```
public class AVLPar extends Par {
  private int h;
  public AVLPar(Object k, Object e, int i)
    {   super(k,e); h = i; }
  public int height() { return h; }
  public int setH(int i)
    {   int g=h; h=i; return g; }
```

V. En digresjon

class BST implements BinTree {
    private **BNode** root;
    void insert(k,e) {
        ....
        = new **BNode**(...)
    ... }
    Position find(k) {
        ...
        BNode p ...
        return p
    }

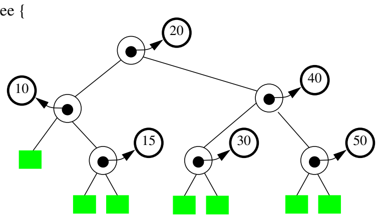


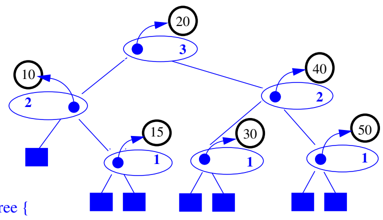
AVLPar



En digresjon

class BST implements BinTree {
    private **BNode** root;
    void insert(k,e) {
        ....
        = new **BNode**(...)
    ... }
    Position find(k) {
        ...
        BNode p ...
        return p
    }





class AVL implements BinTree {
    private **AVLnode** root;
    void insert(k,e) {
        ....
        = new **AVLnode**(...)
    ... }
    Position find(k) {
        ...
        AVLnode p ...
        return p
    }

'Generics'

finnes ikke i JAVA, men de kommer ...

class BST implements BinTree {
    private **BNode** root;
    void insert(k,e) { ....
        = new **BNode**(...)
    ... }
    Position find(k) { ...
        BNode p ...
        return p
    }
    Position parent(Position p) {...}

class **BT[BNode]** implements BinTree {
    private **BNode** root;
    void insert(k,e) { ....
        = new **BNode**(...)
    ... }
    BNode find(k) { ...
        BNode p ...
        return p
    }
    BNode parent(**BNode** v) { ... }

BT bt = new **BT[BNode]**

class **MyNode** extends BNode {
    public String navn();
    MyNode parent() { ... }
    MyNode leftChild() { ... }
... }

BT bst = new **BT[MyNode]**
    MyNode m= bst . root();
    m . navn() ...

class **AVLnode** extends BNode {
    private **AVLnode** left, right, parent;
    public int height() { ... }
    AVLnode parent() { ... }
... }

BT avl = new **BT[AVLnode]**
    avl . leftChild(root()) . height()

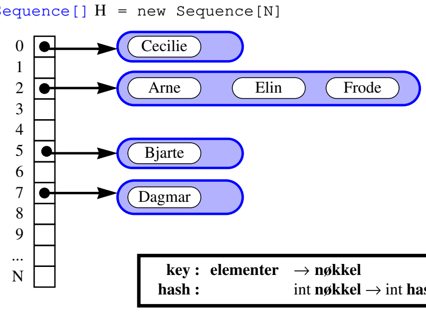
instansiering av en 'generic' BT[**fp**], tilsvarer:
- deklarasjon av BT[**ap**], der **ap** er av type **fp** (subklasse)
- alle forekomster av **fp** er erstattet med **ap**
- operasjoner som returnerer **fp**, vil nå returnere **ap** (kasting blir ofte unødvendig)

VI. Hash tabeller

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

Arne(1972), Bjarte(1975), Cecilie(1970), Dagmar(1977), Elin(1972), Frode(1972)

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



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

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

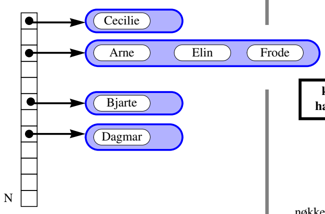
insert(int aa, Object o) { $O(1)$
    H[**hash**(aa)].insertFirst(o) }

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

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

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

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



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

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

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

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

hash er injektiv: hash(k1) = hash(k2) hviss k1 = k2
men nøkkel er det ikke

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 :	**elementer**	→ **persnr**
hash :		int **persnr** → int **siste 4 siffrene**

Arne	123456 12345	2345
Elin	654321 22345	2345
Frode	111111 92345	2345

nøkkel 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 } $O(|S|)$

rem(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

Hash funksjoner

1. Gitt en nøkkel n – et Objekt! – **konverter den til int** k = **k**(n)
2. Gitt en int-representasjon k = **k**(n) – finn en **hashkode hash**(k)
   a) hvis **cp.eq**($n1$,$n2$) 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. bør, men trenger ikke være injektiv, f.eks.

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 : $\boxed{\textbf{hash}(k) = k \textbf{ mod } N < N}$

c) ingen ‘‘beste, generelle’’ løsning :
   unngå konflikter : **hash**(k) = 1 er ikke bra
   minimaliser gjennomsnittlig lengde for hver samling hash(k):
     find(k) = finn k i – samling, sekvens ... **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**: $\boxed{\begin{array}{l}\textbf{hash}(k) = (a * k + b) \textbf{ mod } N \\ \text{der } N \text{ er et primtall, } a > 0,\ b \ge 0\end{array}}$

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**) }