Træer

En meget vigtig datastruktur
Repræsentation af **sprog** (i meget generel betydning), **syntakstræer:**

- Java (i en compiler), SQL (i et databasesystem), XML ....
- Naturligt sprog

Søgetræer:

- Generel repræsentation af (sorterede) mængder og funktioner
- Databasesystemer....

Dagens program:

- Hvad er et træ (datalogisk set)?
- Eksempel på syntakstræer udtrykt vha. nedarvning
  NB: udenfor bogen; find Javafiler via kursets hjemmeside
- Java-stil implementation af generiske træer
  Traversering af træer vha. TreeIterator preorder, postorder, inorder, levelorder
- Binære søgetræer
- Princip og generisk implementation i Java
- Balancerede træer, AVL-træer (+ omtale af et par andre)

Hvad er et træ?

Matematikeren:

- Et specialtilfælde af en graf:
- En sammenhængende skov, hvor en skov er en orienteret, acyklisk graf, med højst én vej mellem to forskellige knuder
- Og så kan vi jo bare bruge kendte repræsentationer for grafer og grafalgoritmer...

Datalogen udnytter naturlig rekursion:

- Et træ består af et stykke data samt nul eller flere træer kaldet deltræer

Træalgoritmer

- Naturligt rekursive

  f.eks. summe træer med tal ...

  $\text{Sum}(t(k,T_1,...,T_n)) = k + \text{Sum}(T_1) + ... + \text{Sum}(T_n)$
- En gang imellem benytte en stak (og sommetider en kø...)

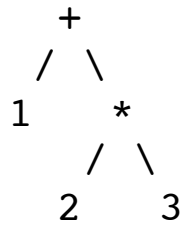
Eksempel på repræsentation af syntaktræer vha. nedarvning

Træer for udtryk defineret ved følgende "abstrakte" grammatik

<udtryk> ::= <tal>
| <udtryk> + <udtryk>
| <udtryk> * <udtryk>

Et *udtrykstræ* er enten

Et *tal-træ*, som indeholder en talkonstant (og 0 deltræer)
Et *plus-træ* har to deltræer, som er udtrykstræer
Et *gange-træ* har to deltræer, som er udtrykstræer



Brug af Javas nedarvning til repræsentation af syntakstræer

```
abstract class ExpressionTree { }

class PlusTree extends ExpressionTree
{ ExpressionTree first; ExpressionTree second;
  public PlusTree(ExpressionTree f, ExpressionTree s) {first=f;second=s;} }

class TimesTree  extends ExpressionTree
{ ExpressionTree first; ExpressionTree second;
  public TimesTree(ExpressionTree f, ExpressionTree s) {first=f;second=s;} }

class NumberTree extends ExpressionTree
{ int value;
  public NumberTree(int v) {value=v;}}
```

(Alternativt: Definere klasse for binære operatorer, og så specialisere den...)

Nedarving udnyttes til at definere et typehierarki — modsat "generiske" træer i Java-stil.

Syntakstræer kan bygges eksplicit

```
ExpressionTree exp  =
        new PlusTree (   new NumberTree(1),
                         new TimesTree(         new NumberTree(2),
                                                new NumberTree(3)));
```

Eller i praksis ved en *parser*, som læser tegn ind fra fil og konstruerer træ efterhånden, enten

- top-down, eller
- bottom-up

Rekursive metoder udnytter også nedarvning....

```
abstract class ExpressionTree
{   public abstract void printStackCode(); }

class PlusTree extends ExpressionTree
{  ExpressionTree first; ExpressionTree second;
   public void printStackCode()
     {first.printStackCode();
      second.printStackCode();
      System.out.println("PLUS");} }

class TimesTree  extends ExpressionTree
{ ExpressionTree first; ExpressionTree second;
  public void printStackCode()
     {first.printStackCode();
      second.printStackCode();
      System.out.println("TIMES");}  }

class NumberTree extends ExpressionTree
{ int value;
  public void printStackCode()
    {System.out.println("PUSH "+value);}  }
```

```
exp.printStackCode();
==>
        PUSH 1
        PUSH 2
        PUSH 3
        TIMES
        PLUS
```

Analogt "post-order traversal" ≈

1. gør noget ved deltræerne, 2. gør noget for denne knude

```
abstract class ExpressionTree
{ public abstract void printExpression(); }

class PlusTree extends ExpressionTree
{   ExpressionTree first; ExpressionTree second;
    public void printExpression()
      {System.out.print("(");
       first.printExpression();
       System.out.print("+");
       second.printExpression();
       System.out.print(")");}  }

class TimesTree  extends ExpressionTree
{  ExpressionTree first; ExpressionTree second;
   public void printExpression()
     {System.out.print("(");
      first.printExpression();
      System.out.print("*");
      second.printExpression();
      System.out.print(")");}   }

class NumberTree extends ExpressionTree
{  int value;
   public void printExpression()
    {System.out.print(value);}   }
```

```
exp.printExpression();
==> (1+(2*3))
```

Ser vi bort fra udskrift af parenteserne: Analogt "in-order traversal" ≈

1. Gør noget ved venstre deltræ,
2. Gør noget for denne knude,
3. Gør noget ved højre deltræ

```
abstract class ExpressionTree
{ public abstract void printLispCode(); }

class PlusTree extends ExpressionTree
{   ExpressionTree first; ExpressionTree second;
    public void printLispCode()
      {System.out.print("(PLUS");
       first.printLispCode();
       second.printLispCode();
       System.out.print(")");}  }

class TimesTree  extends ExpressionTree
{   ExpressionTree first; ExpressionTree second;
    public void printLispCode()
      {System.out.print("(TIMES");
       first.printLispCode();
       second.printLispCode();
       System.out.print(")");}  }

class NumberTree extends ExpressionTree
{   int value;
   public void printLispCode()
     {System.out.print(" "+value);}  }
```

```
exp.printLispCode ();
==> (PLUS 1(TIMES 2 3))
```

Ser vi bort fra udskrift af parenteserne: Analogt "pre-order traversal" ≈

1 Gør noget for denne knude,
2 Gør noget ved deltræerne

Repræsentation af Java-stil generiske binære træer

```
public class BinaryNode {
   // konstruktorer og de metoder kan kan forvente
    private Object     element;
    private BinaryNode left;
    private BinaryNode right;   }

public class BinaryTree
{ // konstruktorer og de metoder kan kan forvente
   private BinaryNode root;   }

abstract class TreeIterator
{ public TreeIterator( BinaryTree theTree ) {t=theTree;current=null;}
  // den slags metoder som hører til en iterator...
  // blandet abstract og def. vha. abstract
  protected BinaryTree t;
  protected BinaryNode current;  }
```

class PostOrder extends TreeIterator
{ //*definition af abstrakte metoder fra TreeIterator*
  ...
  //*internt snask:*
  protected static class StNode { }
  protected Stack s; }

class InOrder extends PostOrder { } // *usmukt genbrug :(*

class PreOrder extends TreeIterator { *... også noget med en stak* }

class LevelOrder extends TreeIterator { *... noget med en kø*}

```
abstract class TreeIterator {

  public TreeIterator(BinaryTree theTree) {t = theTree; current = null;}

  abstract public void first( );

  final public boolean isValid( ) {return current != null;}

  final public Object retrieve( ) {
        if( current == null ) throw ... ;
        return current.getElement( ); }

  abstract public void advance( );

  protected BinaryTree t;
  protected BinaryNode current;  }
```

class PostOrder extends TreeIterator {...}

Hvis Java havde haft ægte co-rutiner kunne det implementeres ved rekursion

Pseudokode:

```
class PostOrder extends TreeIterator {...}
   public void first {postOrderTraverse(t.getRoot());}

   private void postOrderTraverse(BinaryNode t); {
      if(t==null) return;
      postOrderTraverse(t.getLeft());
      postOrderTraverse(t.getRight());
      current=t;
      detach();
      return;}

  public void advance( ); {resume()}   }
```

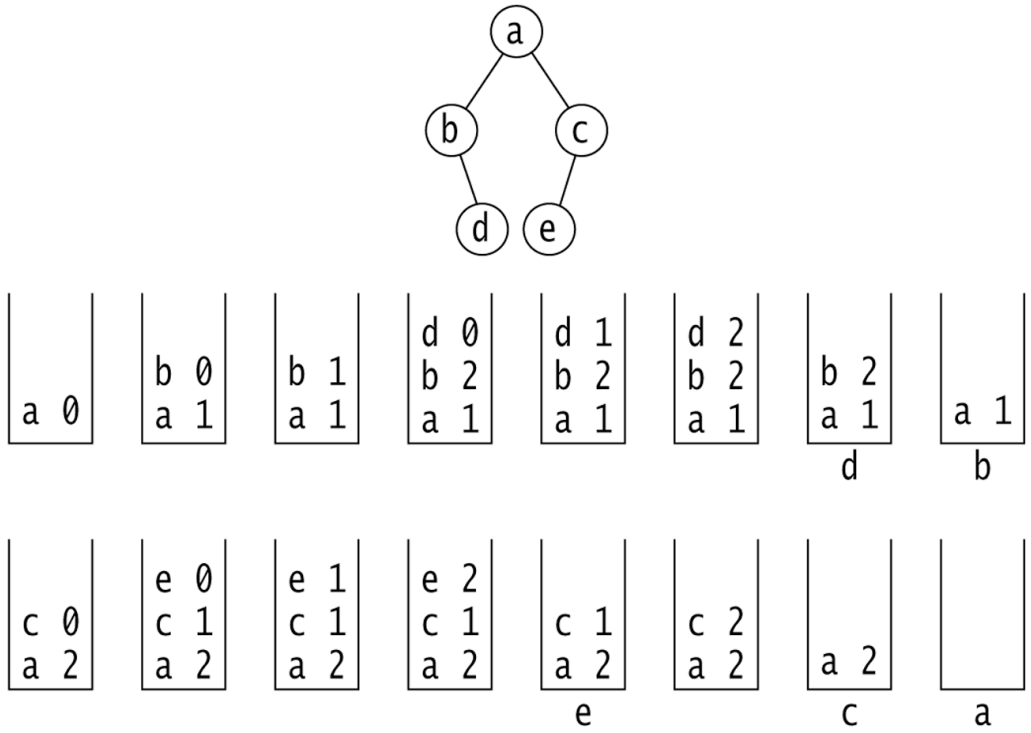
Primitiver a la Simula68 (ikke i Java)

detach() — læg metode til at sove indtil der kommer en resume()
resume() — start en detach'et proces

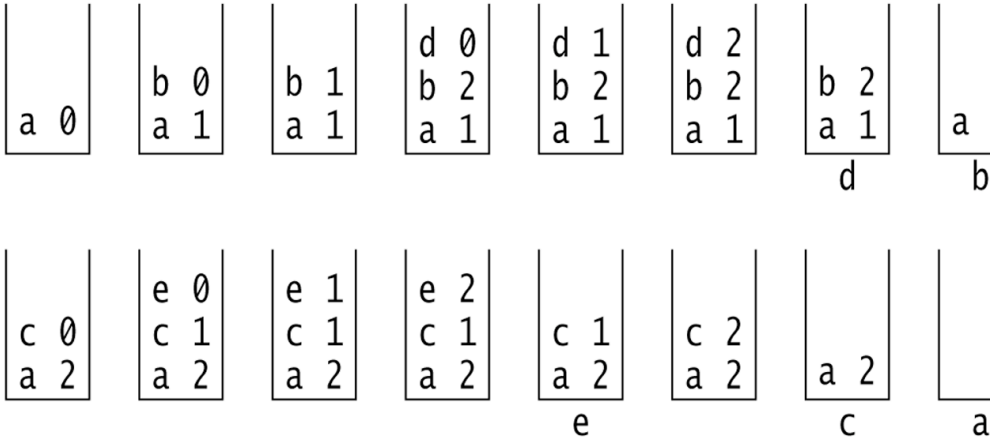
OBS: Preorder og inorder fås ved at bytte rundt linjerne i xxxOrderTraverse()

Disse primitiver findes ikke i Java, så derfor programmeres med eksplicit stak.

```
class PostOrder extends TreeIterator {
 // Intern stak i mangel på rekursion:

 protected static class StNode
    { BinaryNode node;
      int timesPopped;
      StNode( BinaryNode n )
       {node = n; timesPopped = 0;}}

protected Stack s;

.... }
```



```
class PostOrder extends TreeIterator {
 // Intern stak i mangel på rekursion:

 protected static class StNode
    { BinaryNode node;
      int timesPopped;
      StNode( BinaryNode n )
       {node = n; timesPopped = 0;}}

protected Stack s;

public PostOrder( BinaryTree theTree )
    {super(theTree); s=new ArrayStack();
     s.push(new StNode(t.getRoot()));}

  public void first( )
    {s.makeEmpty( );
     if( t.getRoot( ) != null ) {s.push(new StNode(t.getRoot())); advance();}}
```




```
public void advance( )
  {if( s.isEmpty( ) )
     {if( current == null ) throw ...;
       current = null; return; }
    StNode cnode;
    for( ; ; )
      {cnode = ( StNode ) s.topAndPop( );
       if( ++cnode.timesPopped == 3 ) {current = cnode.node; return;}
       s.push( cnode );
       if( cnode.timesPopped == 1 )
         { if(cnode.node.getLeft() !=null)
                        s.push(new StNode(cnode.node.getLeft()));}
        else  // cnode.timesPopped == 2
       {if(cnode.node.getRight()!=null)
                    s.push(newStNode(cnode.node.getRight()));}}}
}
```

//NB: der stilles ingen eksamensspørgsmål om hvordan den stak bruges

// (læreren synes det er noget knoldkode)

Binært *søge*træ

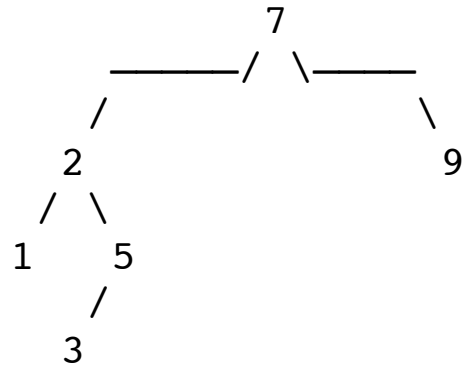
En måde at opbevare og vedligeholde en samling objekter på.

- objekterne holdes sorterede
  - dvs. at finde objekter ≈ log n
- indsættelse af nye objekter kræver ikke ny sortering eller flytning af n/2 elementer
- indsættelse med "inkrementel sortering" ≈ log n

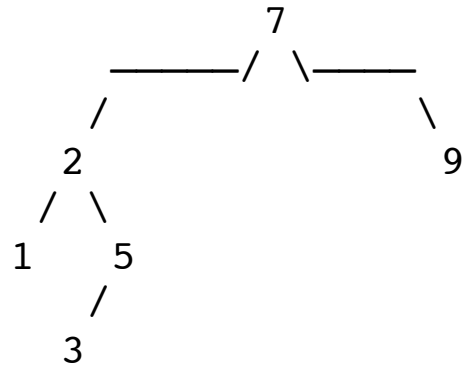
Definition: Et *binært søgetræ* er et binært træ hvor hver knude holder et element af en mængde udstyret med "<" og "=", således at

alle værdier i venstre deltræ < værdi på knude < alle værdier i højre deltræ

Eksempel



Eksempel



Egenskab: Hvis et binært søgetræ udskrives "in-order" kommer elementerne ud i voksende rækkefølge

Skitse af definition: En binært træ er *balanceret* hvis det er ca. lige dybt over det hele.

Egenskab: Et balanceret, binært træ med n elementer har højde ≈ log n.
Dvs. søgning i balanceret, binært søgetræ er O(log n).

Repræsentation af Binære søgetræer i Java-stil

Upraktisk at specialisere fra BinaryTree, da Object'erne skal være Comparable + lukke for modifikationer.

class BinaryNode

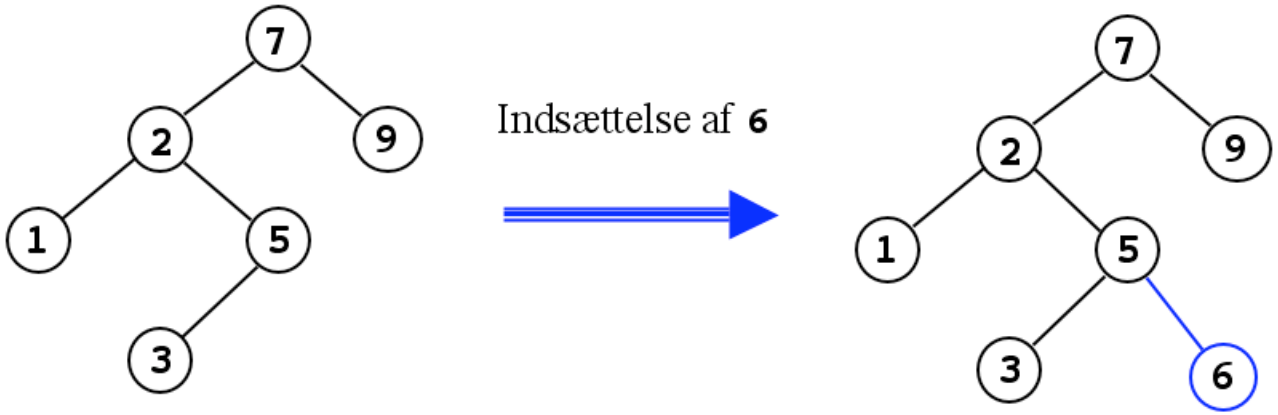
```
{ BinaryNode(Comparable theElement){element=theElement;left=right=null;}
  Comparable element; BinaryNode left; BinaryNode right; }
```

public class BinarySearchTree

```
{ public BinarySearchTree( ) {root = null; }
 // diverse nyttige metoder, bl.a.

  public void insert (Comparable x) { halv-kompliceret stads }

  public Comparable find( Comparable x ) // returnerer x hvis den findes;ellers null
    { return elementAt( find( x, root ) );}

  private Comparable elementAt( BinaryNode t ) {return t == null ? null : t.element;}

  private BinaryNode find( Comparable x, BinaryNode t )
    { while( t != null )  { if( x.compareTo( t.element ) < 0 ) t = t.left;
                            else if( x.compareTo( t.element ) > 0 ) t = t.right;
                            else return t; }
      return null; }

  protected BinaryNode root;  }
```

Indsættelse af element x ≈

At lede efter elementet x, og hvis det ikke findes, indsætte det, hvor det burde have været:

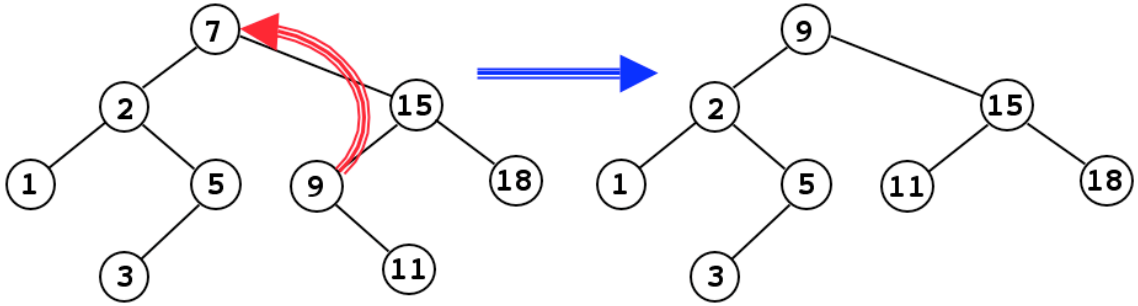


I Java: Metode i BinarySearchTree:

```
public void insert(Comparable x) { root = insert(x, root); }

BinaryNode insert(Comparable x, BinaryNode t) {
   if (t == null) t = new BinaryNode(x, null, null);
   else if (x.compareTo(t.element) < 0) t = insert(x, t.left);
   else if (x.compareTo(t.element) > 0) t = insert(x, t.right);
   else throw new DuplicateItemException(x.toString());
   return t;  }
```

Sletning af element x i binært søgetræ

Bemærk: Problemet kan altid reduceres til at fjerne x i position som rod af et træ (det undertræ, som har x i toppen).



Erstat roden med den knude, der er mindst i rodens højre undertræ.
Denne knude befinder sig længst til venstre i rodens højre undertræ.
Hvis det højre undertræ er tomt, fjernes roden blot fra træet.

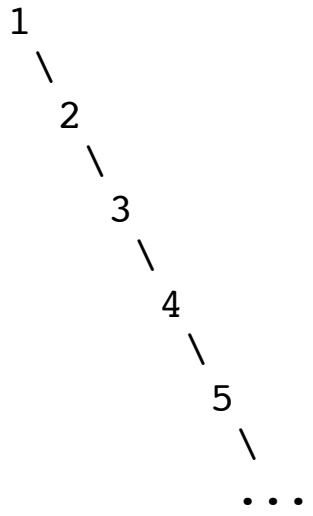
I Java:

```
public void remove(Comparable x) {root = remove(x, root);}

protected BinaryNode remove( Comparable x, BinaryNode t )
   {if( t == null ) throw new ItemNotFoundException( x.toString( ) );
     if( x.compareTo( t.element ) < 0 ) t.left = remove( x, t.left );
     else if( x.compareTo( t.element ) > 0 ) t.right = remove( x, t.right );
     else if( t.left != null && t.right != null ) // Two children
       { t.element = findMin( t.right ).element;
         t.right = removeMin( t.right ); }
      else t = ( t.left != null ) ? t.left : t.right;
      return t; }
```

findMin ≈ søg til venstre
removeMin ≈ søg rekursivt til venstre, og da minimum element ikke har to undertræer, kan træet "trækkes sammen"

Dette er altsammen meget godt, *men* det er kun interessant hvis indsættelser og sletninger ankommer således passende at træet forbliver balanceret.

Eksempel: Værdier indsættes i voksende rækkefølge:



Træet degenererer til en hægtet liste. O(log n) ==> O(n) :(

Altså: Indsættelse og sletning skal justere på træet så det er nogenlunde balanceret.

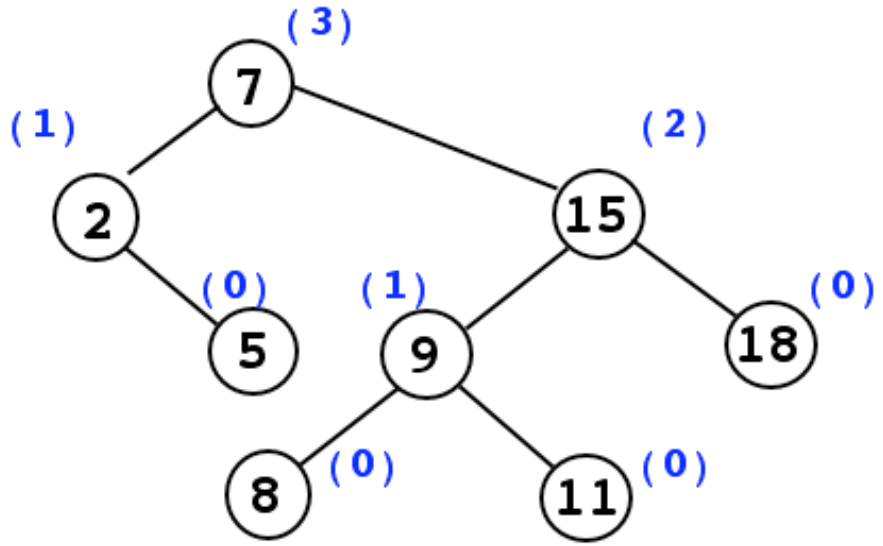
Bogen omtaler forskellige strategier (og der er mange andre beskrevet i litteraturen)

Vi nøjes med at skitsere AVL-træer!

AVL-træer (Adelson-Velskii og Landis, 1962)

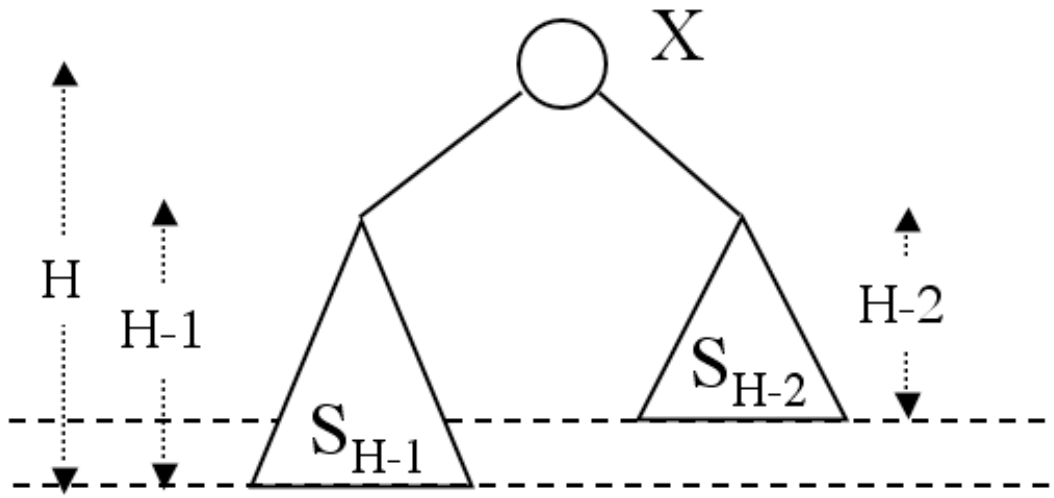
For enhver knude i træet gælder, at højden af dens venstre undertræ og højden af dens højre undertræ højst afviger med 1.

Eksempel på AVL-træ:



Det kan bevises, at denne egenskab er tilstrækkelig for at sikre log n dybde.

Skitse af metode for indsættelse i AVL-træ

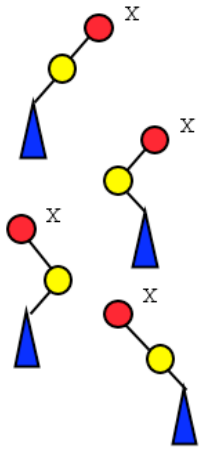


Indsættelse i X's venstre undertræ kan ødelægge AVL-egenskaben.

I det følgende betegner X den "dybeste" knude, som er rod i et deltræ, der ikke længere opfylder AVL-egenskaben.

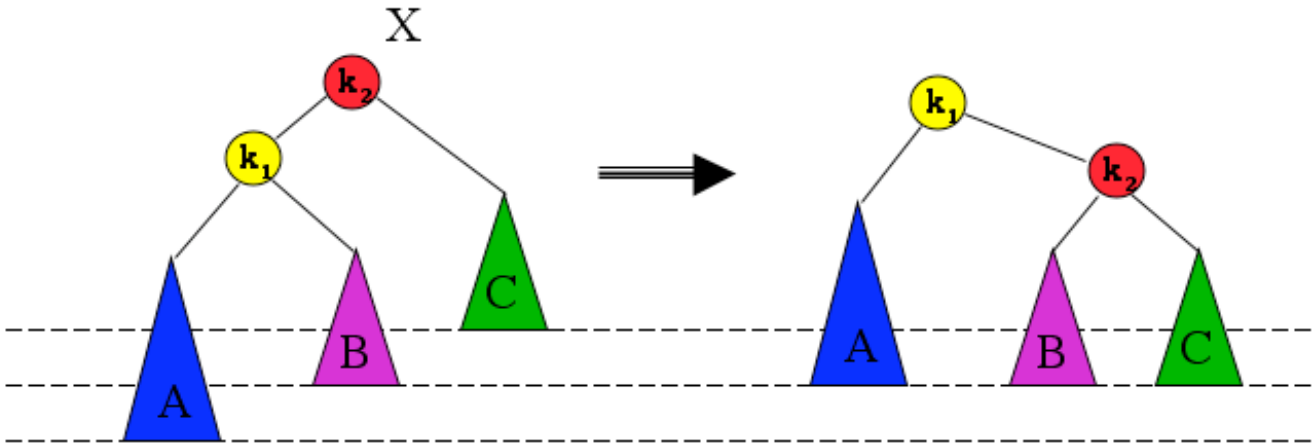
Ved indsættelse i et af X's undertræer er der 4 mulige tilfælde:

1. Indsættelsen sker i det *venstre* undertræ af X's *venstre* søn.
2. Indsættelsen sker i det *højre* undertræ af X's *venstre* søn.
3. Indsættelsen sker i det *venstre* undertræ af X's *højre* søn.
4. Indsættelsen sker i det *højre* undertræ af X's *højre* søn.

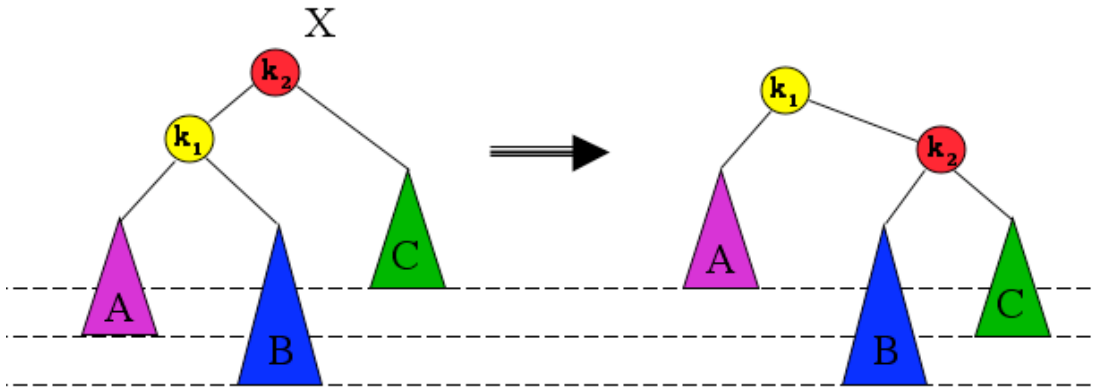


Tilfælde 1 og 4 er symmetriske.
Tilfælde 2 og 3 er symmetriske.

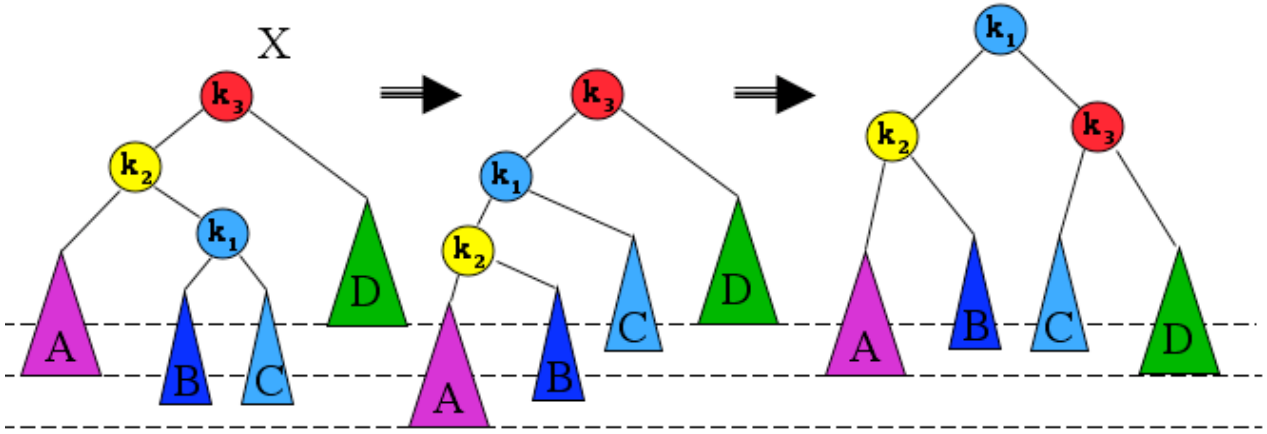
Tilfælde 1: En højrerotation genskaber balancen



Tilfælde 2: En højrerotation genskaber ikke balancen



Der skal en dobbeltrotation til:



Først rotere til venstre i venstre deltræ; dernæst til højre i hele træet!
Vi skåner tilhøreren (og læreren!) for implementationen i Java!

Afslutning:

Træer — vigtig datastruktur til at repræsentere "af natur træagtige ting"

- syntakstræer
- katalogstrukturer
- procestræer i operativsystemer
- ....

(Binære) søgetræer:

- Vigtig datastrukturer til effektiv lagring og genfinding.
- Essentielt: At holde træerne balancerede!
- Databasesystemer er fyldt med træer og balanceringsalgoritmer
- ....

og hashing som vi kommer til senere...