Hoofdstuk 6

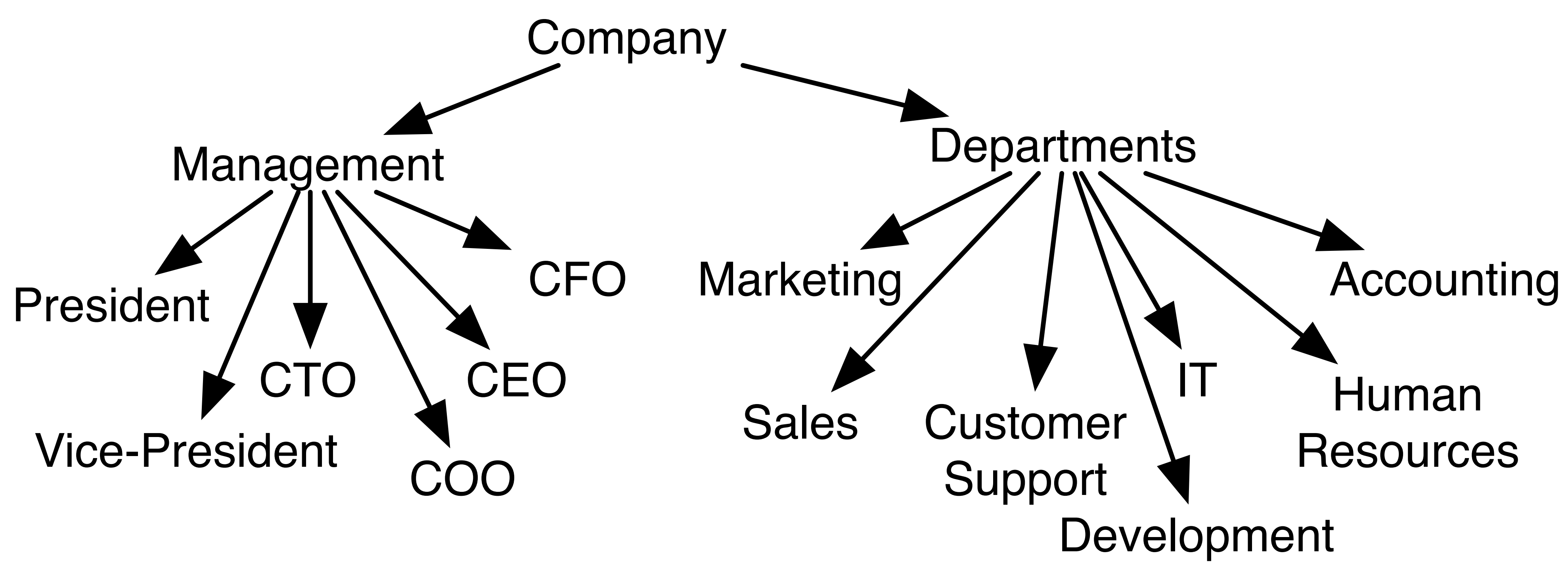
Bomen

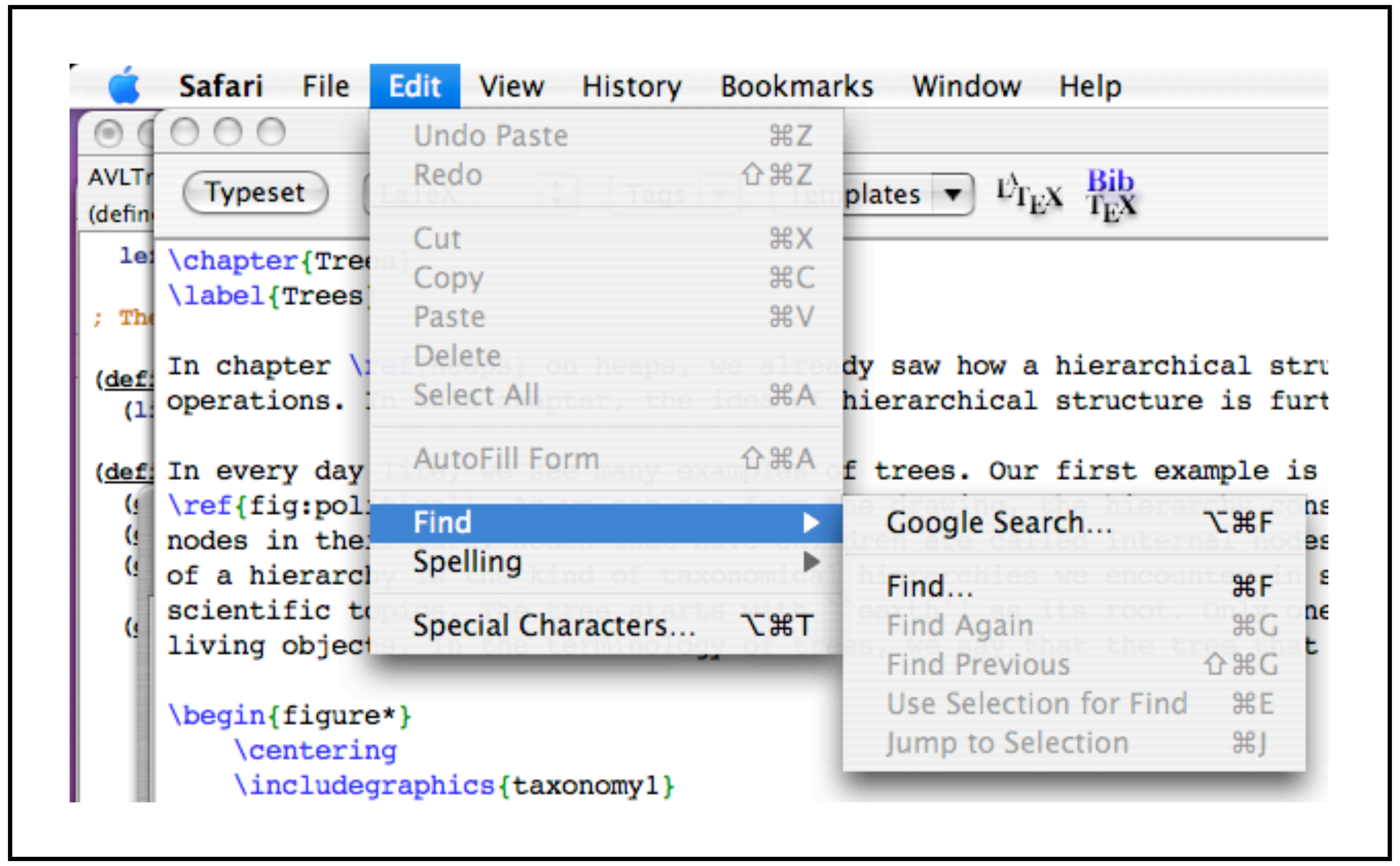
Inhoud

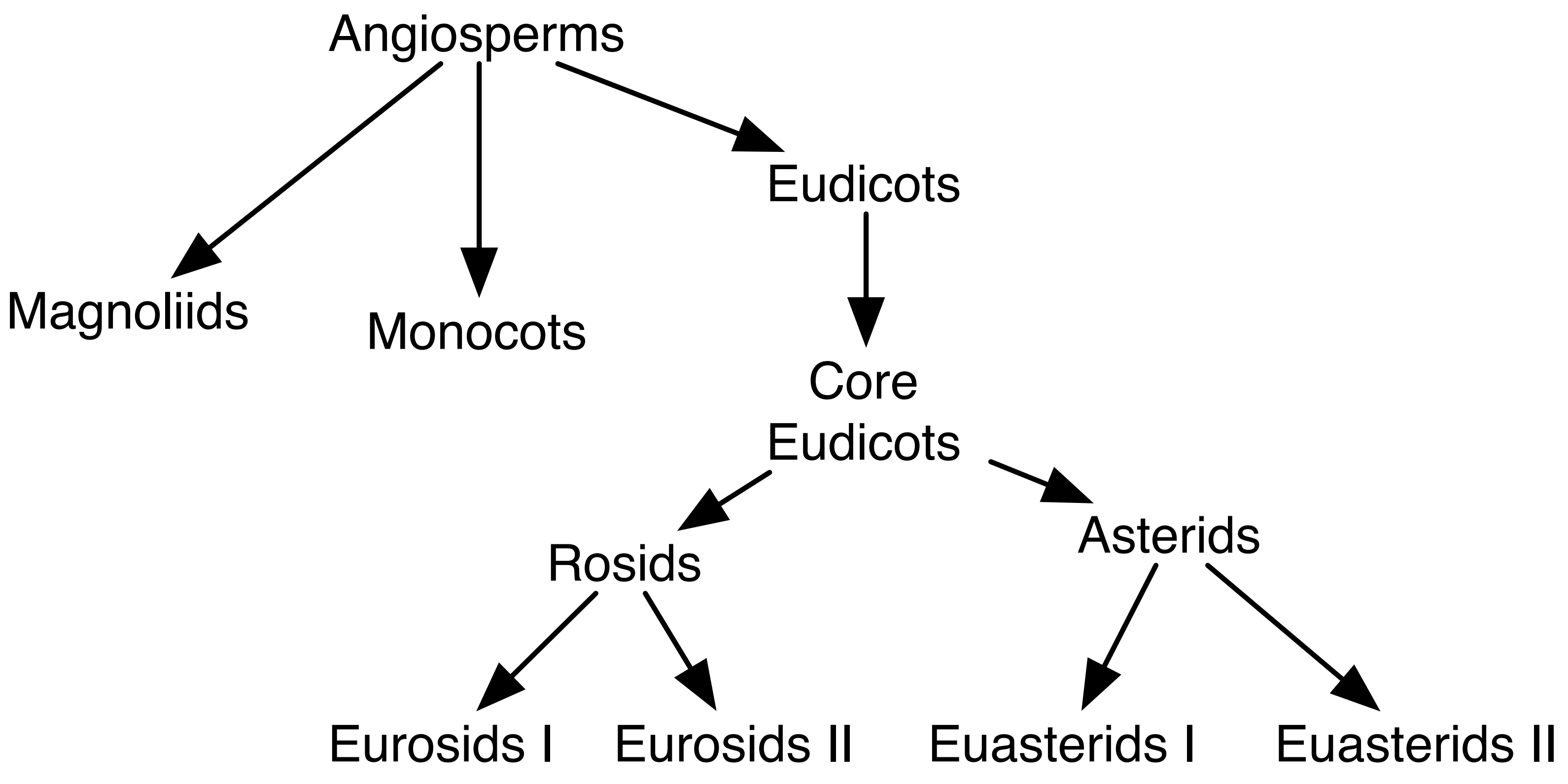
1. Structuur van bomen
2. Doorlopen van bomen
3. Binaire zoekbomen
4. AVL bomen

6.1 Structuur van bomen

Voorbeelden van Hiërarchieën





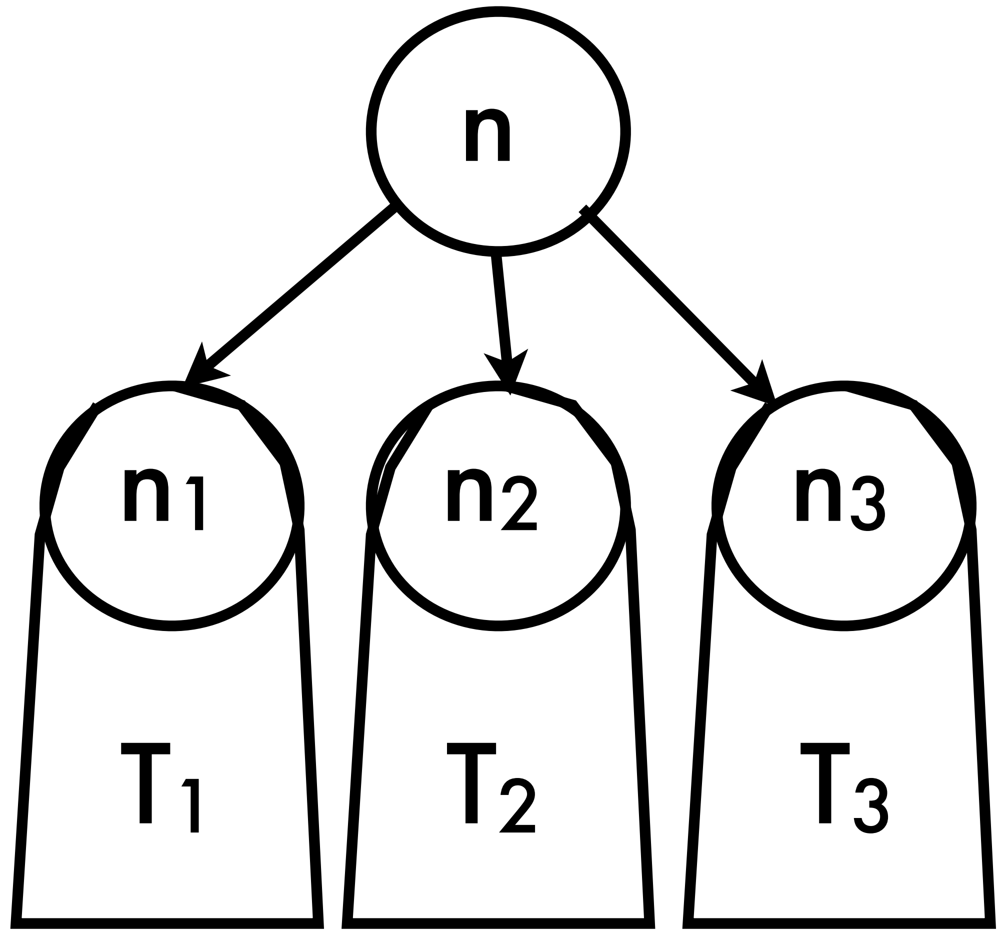


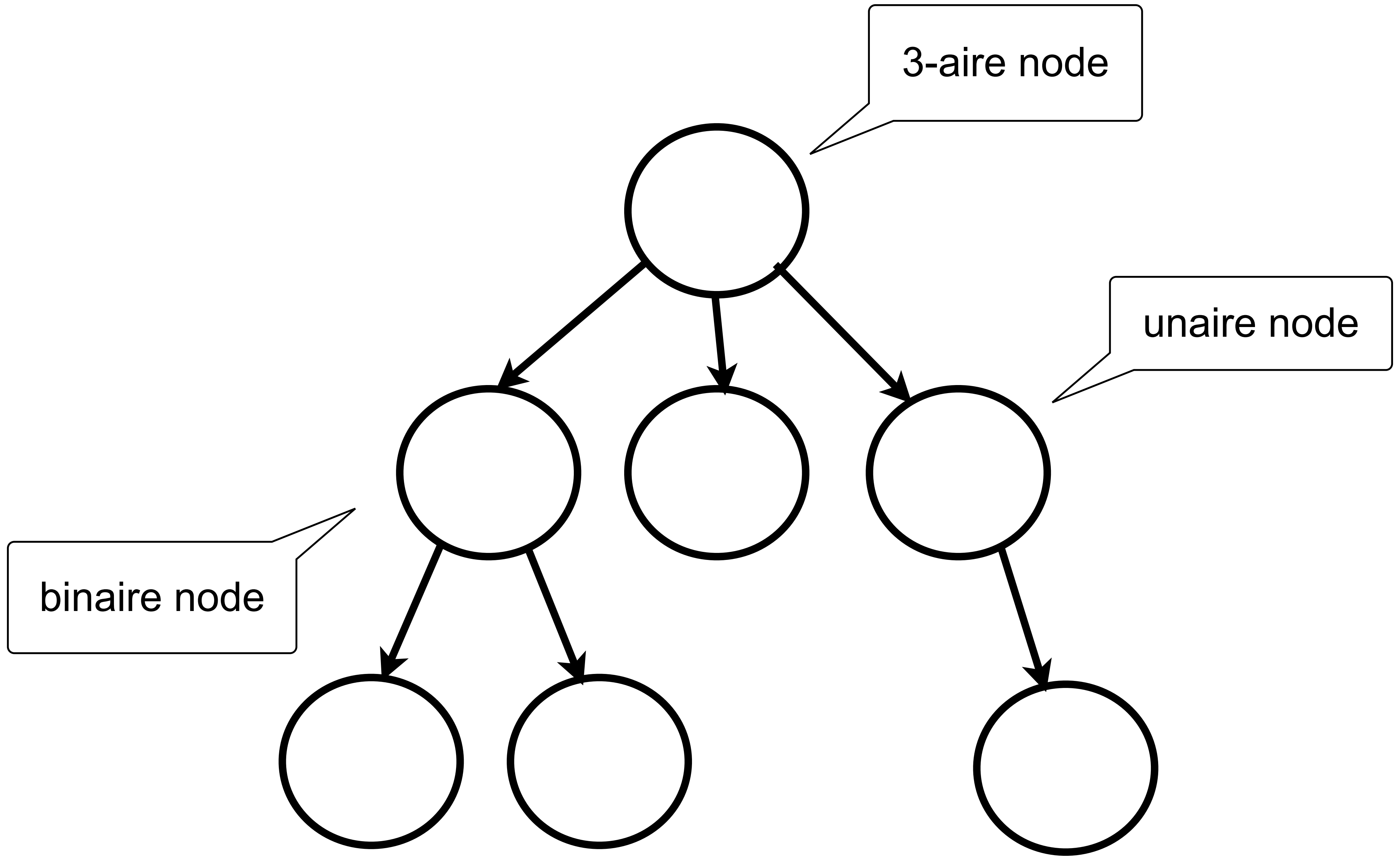
Bomen: Terminologie

De lege boom is een boom

Elke niet lege boom bestaat uit nodes. Eén node heet de root. Iedere node heeft een verwijzing naar een aantal kinderen. Er zijn geen lussen.

Elke node heeft een aantal subbomen





De ariteit van een node is het aantal kinderen

k-aire nodes

Complete k-aire bomen

Een complete boom heeft geen gaten wanneer we hem van links naar rechts lezen, laag per laag

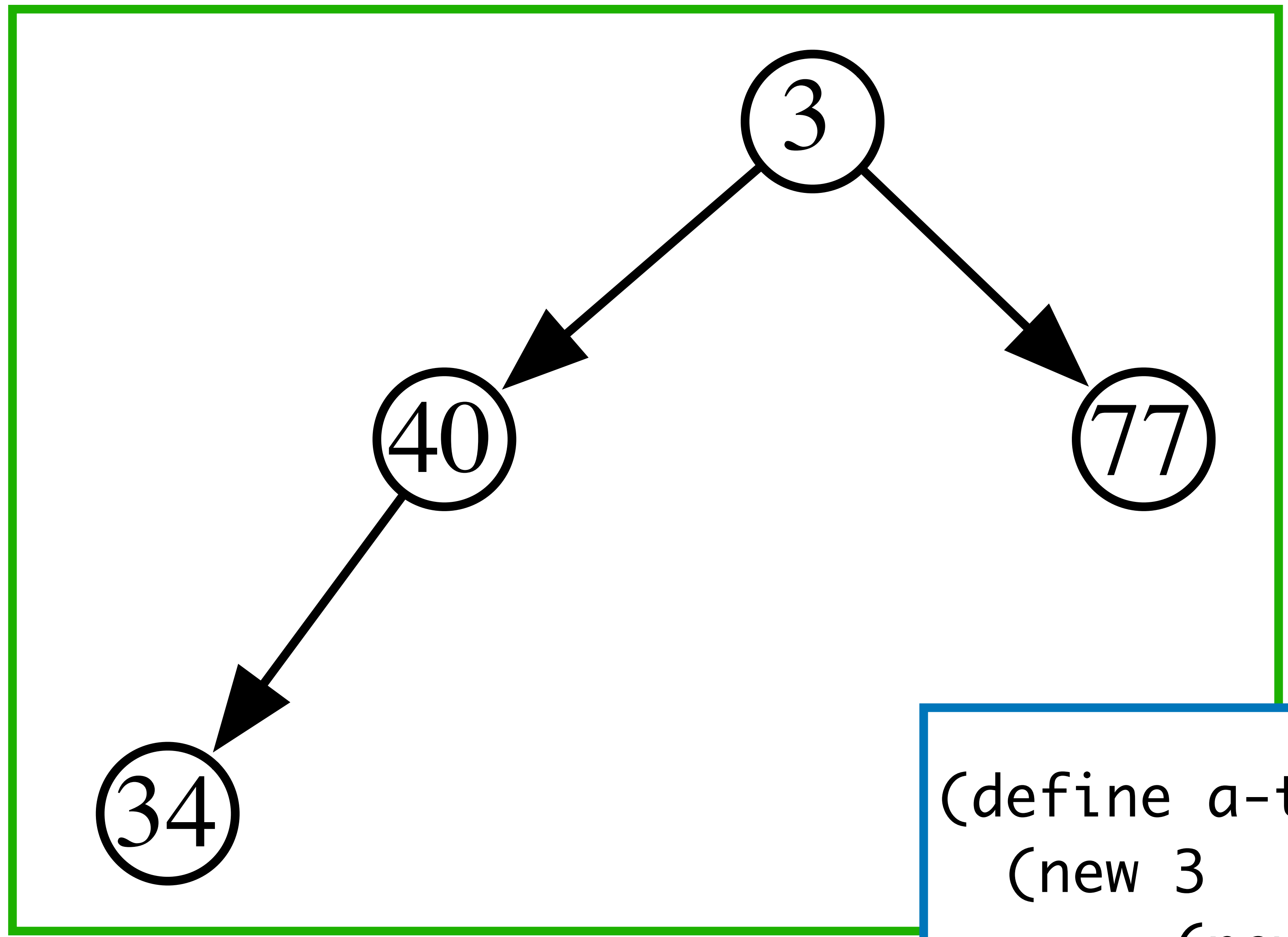
De hoogte van een k-aire complete boom is $\log_k(n)$

Met andere woorden: niet-complete bomen zijn **minstens zo hoog**.

Binaire bomen (k = 2)

```
ADT binary-tree

null-tree
  binary-tree
new
  ( any binary-tree binary-tree → binary-tree )
null-tree?
  ( binary-tree → boolean )
left
  ( binary-tree → binary-tree )
left!
  ( binary-tree binary-tree → binary-tree )
right
  ( binary-tree → binary-tree )
right!
  ( binary-tree binary-tree → binary-tree )
value
  ( binary-tree → any )
value!
  ( binary-tree any → binary-tree )
```

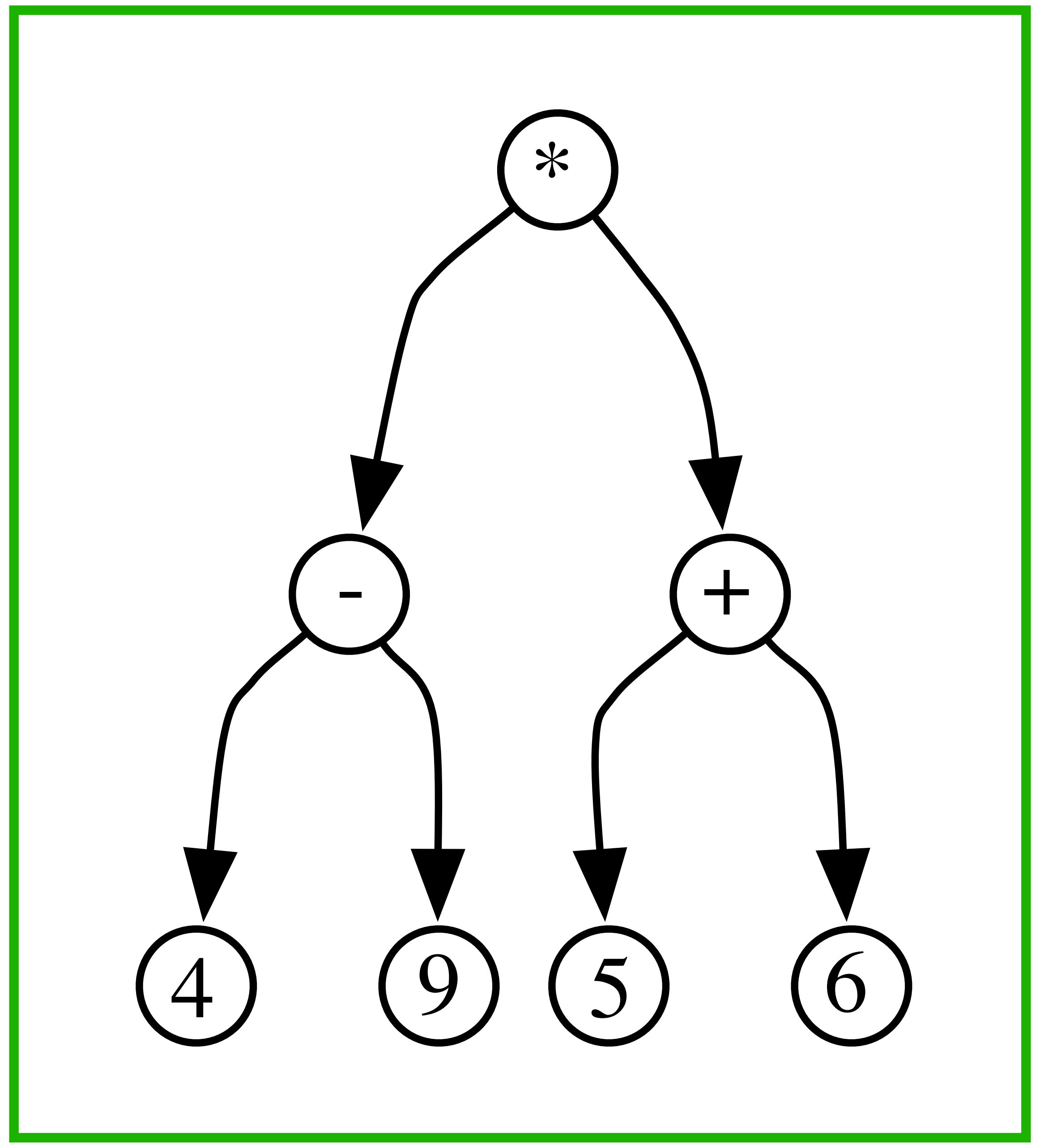
Voorbeeld




```
(define a-tree
  (new 3
       (new 40
            (new 34
                 null-tree
                 null-tree)
            null-tree)
       (new 77
            null-tree
            null-tree)))
```

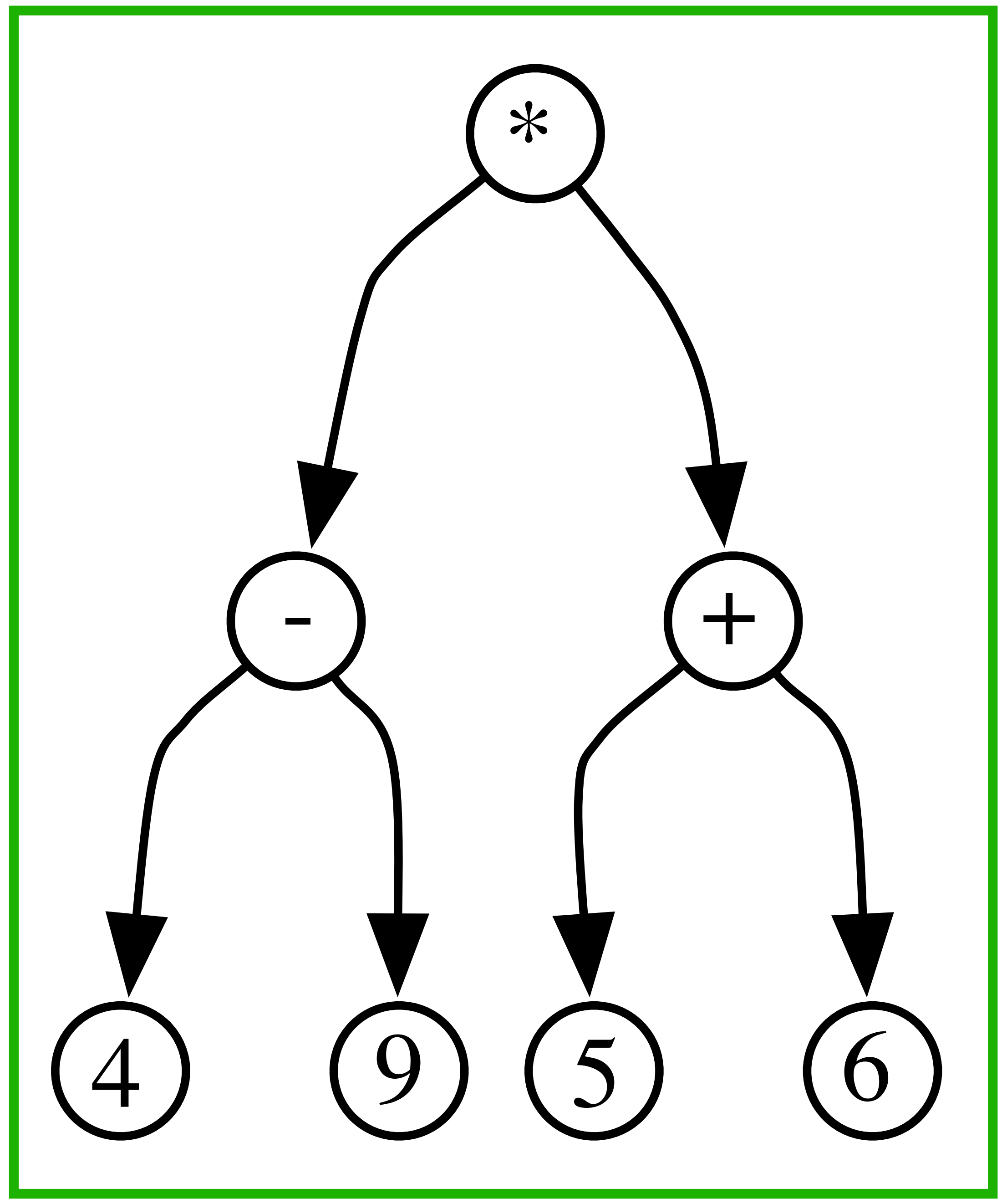
Voorbeeld: Expressiebomen

```
(define t4 (new 4 null-tree null-tree))

(define t9 (new 9 null-tree null-tree))

(define t5 (new 5 null-tree null-tree))

(define t6 (new 6 null-tree null-tree))

(define minus (new '- t4 t9))

(define plus (new '+ t5 t6))

(define times (new '* minus plus))
```



Voordeel van Boomstructuren: Recursie

```
(define (eval tree)
  (cond ((eq? (value tree) '+)
         (+ (eval (left tree))
            (eval (right tree))))
        ((eq? (value tree) '*)
         (* (eval (left tree))
            (eval (right tree))))
        ((eq? (value tree) '-)
         (- (eval (left tree))
            (eval (right tree))))
        (else (value tree))))
```

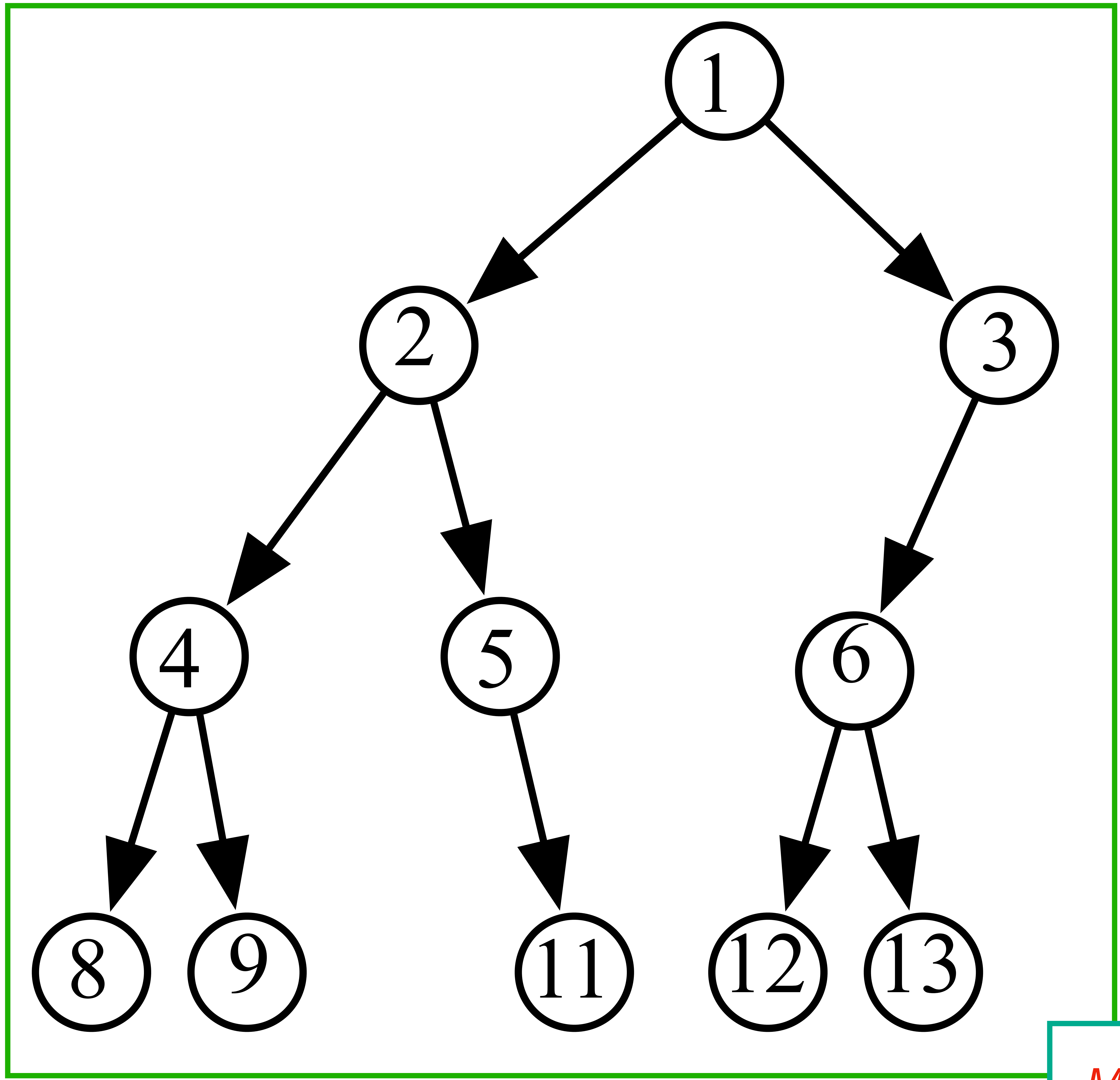


Hierarchische structuren hebben veel structuur en zijn algoritmisch dus heel aantrekkelijk omdat we recursie kunnen uitbuiten.

Gelinkte Representatie

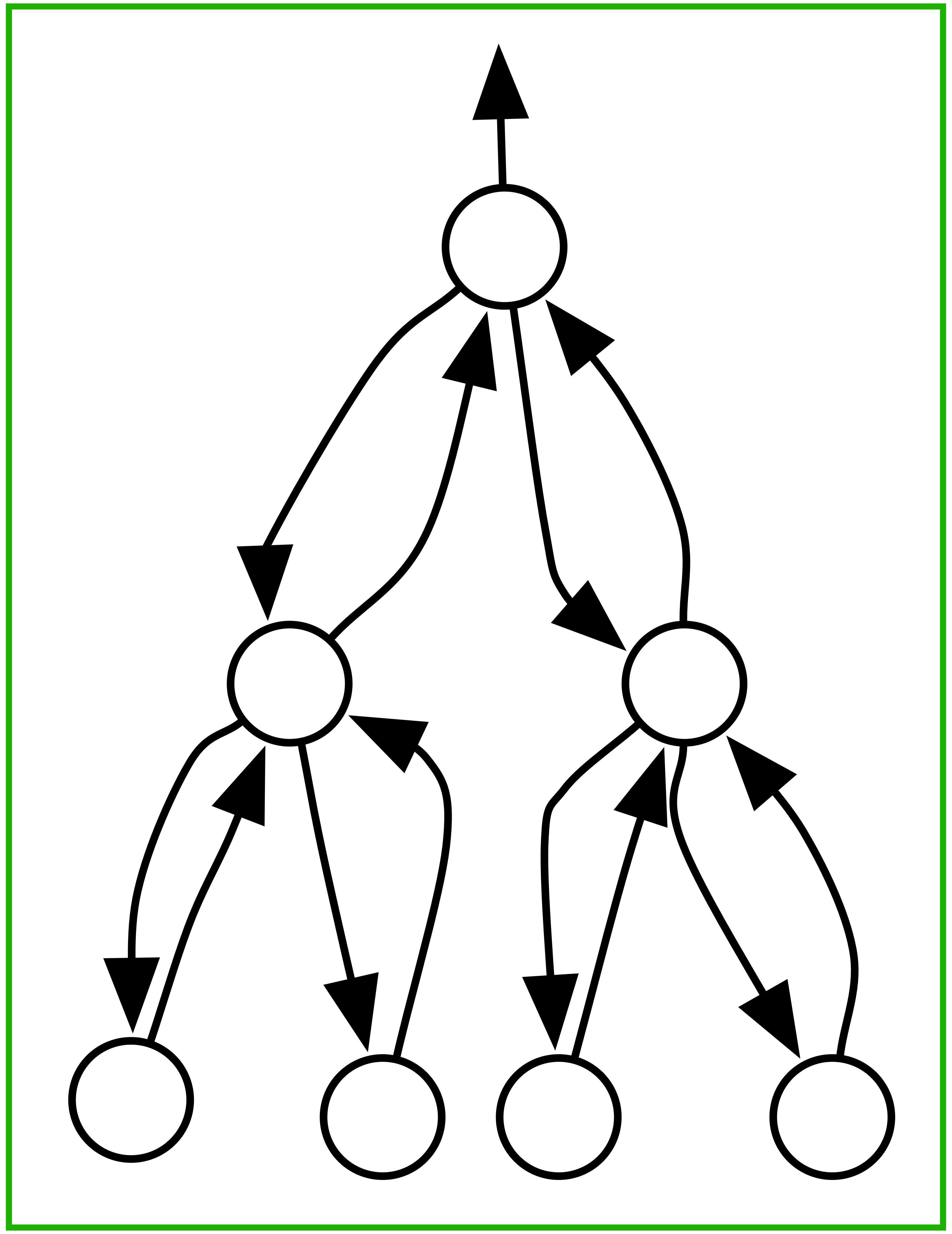
```
(define-record-type tree
  (new v l r)
  tree?
  (v value value!)
  (l left left!)
  (r right right!))

(define null-tree '())

(define (null-tree? node)
  (eq? node null-tree))
```

Alternatief: Vectoriële Representatie



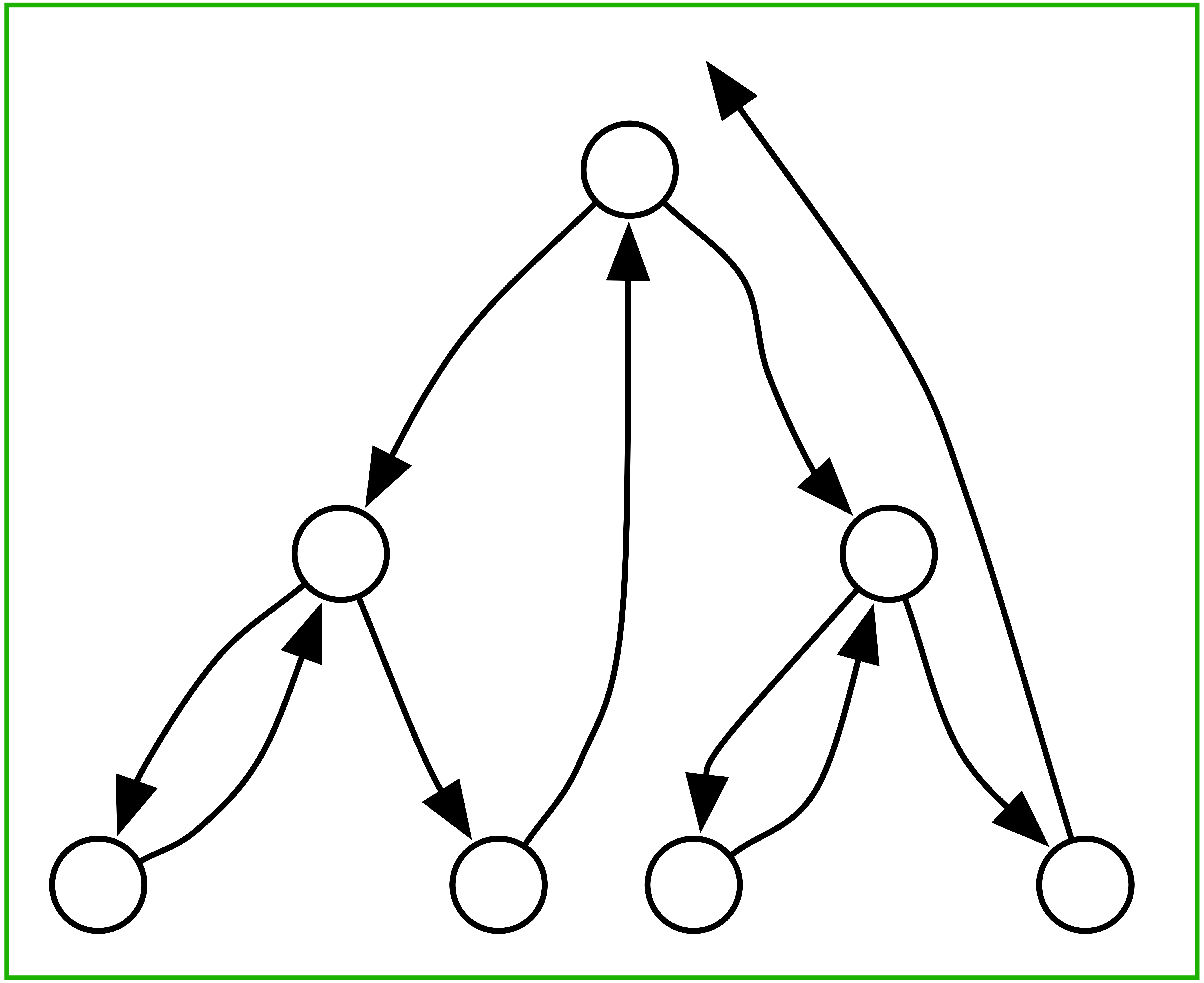
Mind the gaps!

Dubbelgelinkte Bomen



Vader van een knoop vastpakken wordt makkelijker

Threaded Trees



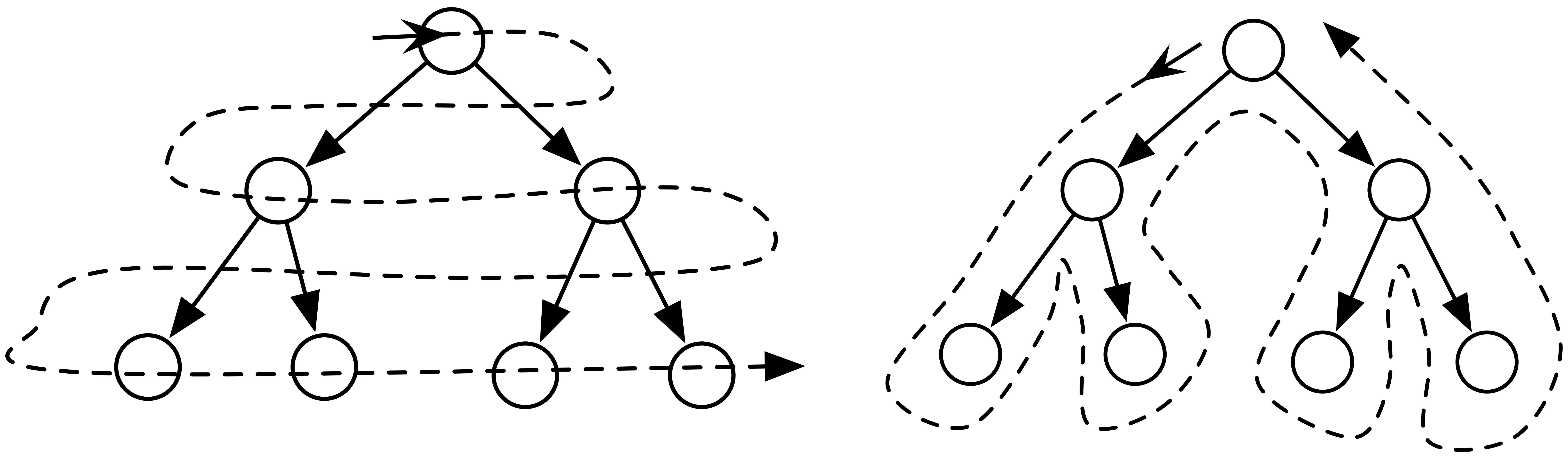
Gebruik ongebruikte “right” ‘() pointers voor navigatie

6.2 Doorlopen van bomen

Doorlopen van Bomen

De volgorde van `map`*,* `for-each` *op lijsten is natuurlijk: van links naar rechts. Voor bomen is de volgorde minder voor de hand liggend.*

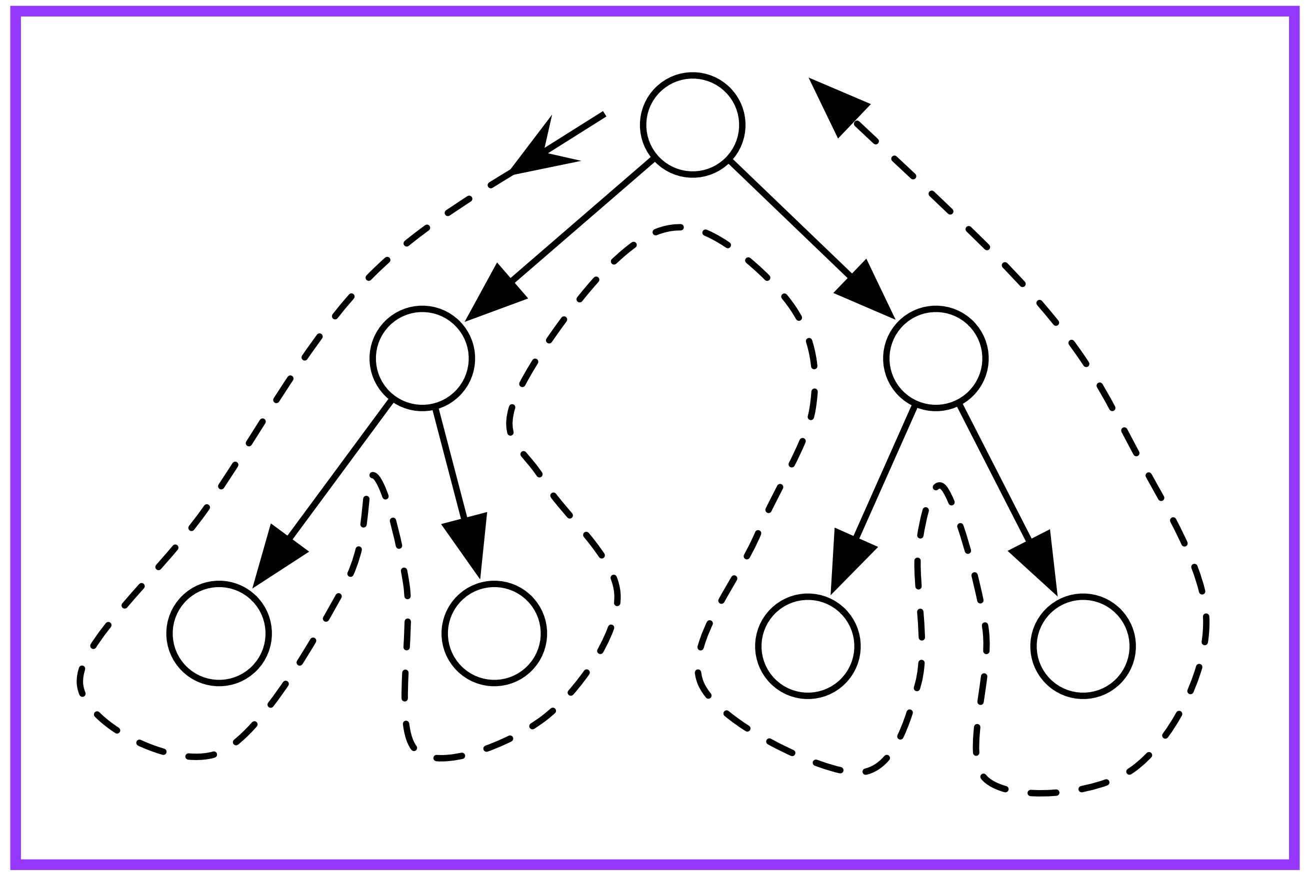
Voor bomen: 2 fundamenteel verschillende families van algoritmen

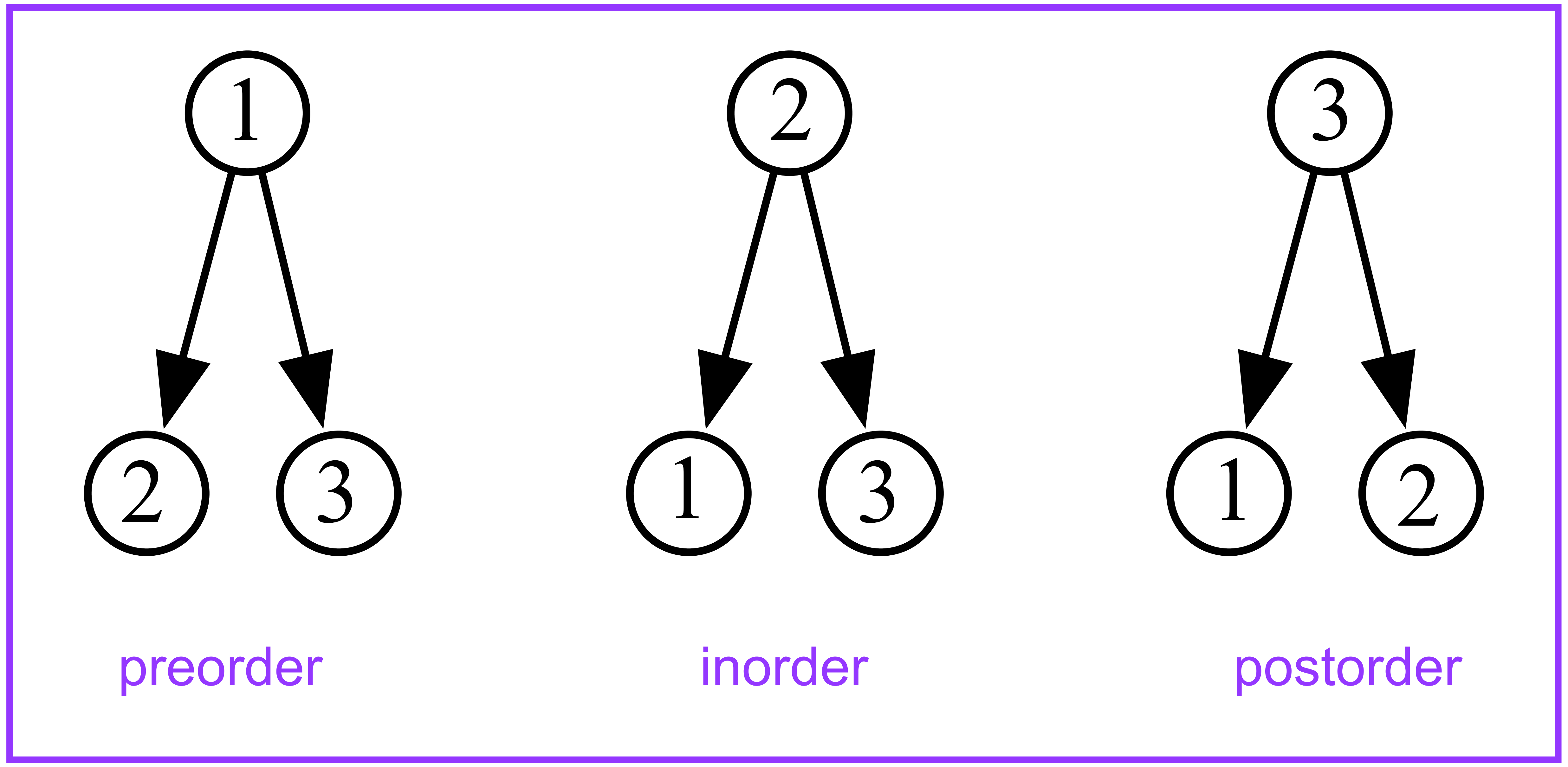


breadth first traversal

depth first traversal

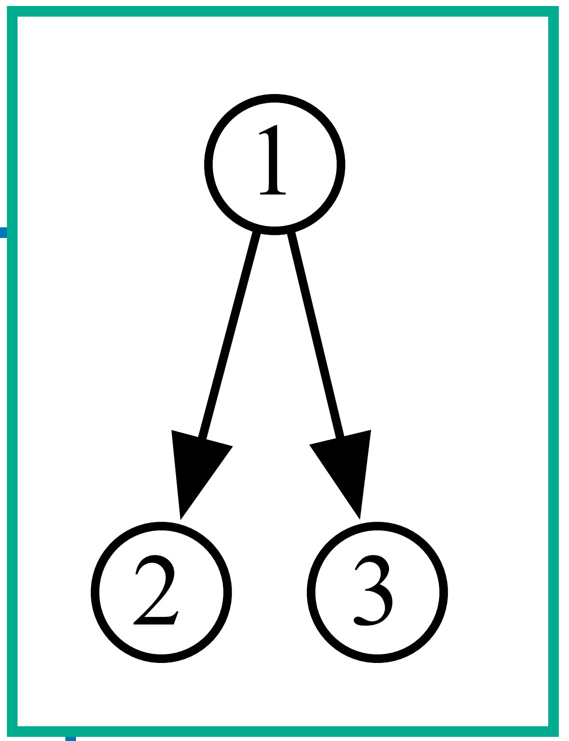
Depth-First Traversals: Verdere Opsplitsing



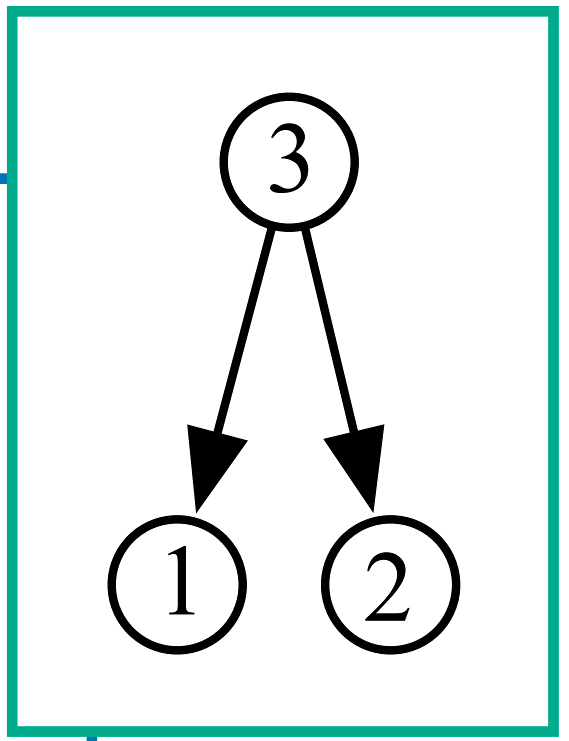


Recursieve Implementaties

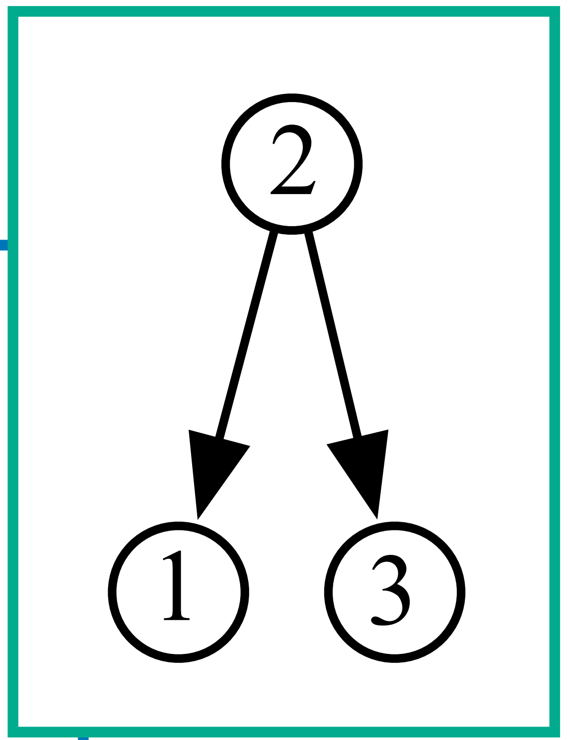
```
(define (pre-order tree proc)
  (define (do-traverse current)
    (when (not (null-tree? current))
      (proc (value current))
      (do-traverse (left current))
      (do-traverse (right current))))
  (do-traverse tree))
```




```
(define (post-order tree proc)
  (define (do-traverse current)
    (when (not (null-tree? current))
      (do-traverse (left current))
      (do-traverse (right current))
      (proc (value current))))
  (do-traverse tree))
```




```
(define (in-order tree proc)
  (define (do-traverse current)
    (when (not (null-tree? current))
      (do-traverse (left current))
      (proc (value current))
      (do-traverse (right current))))
  (do-traverse tree))
```

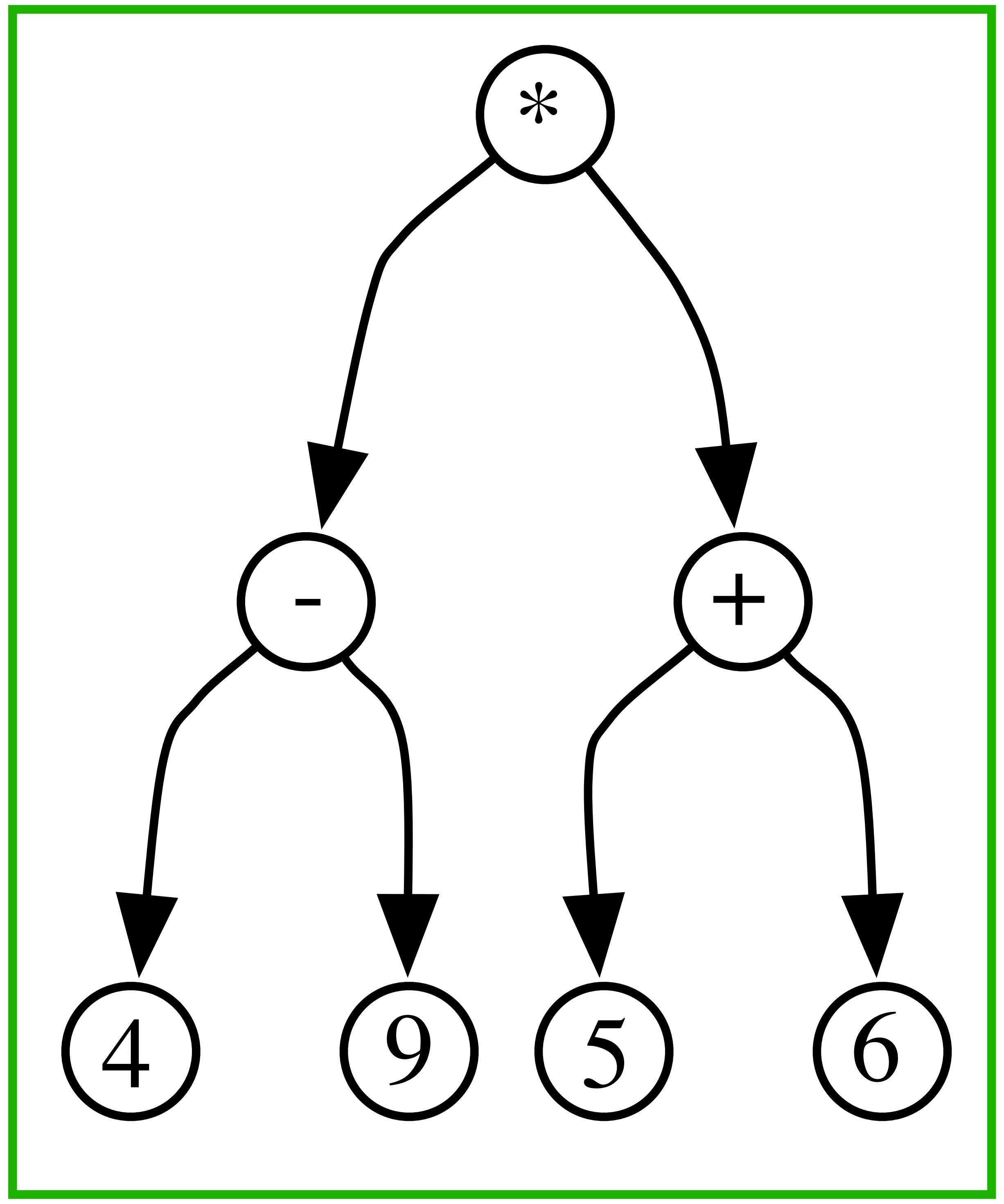


Traversals & Expressiebomen

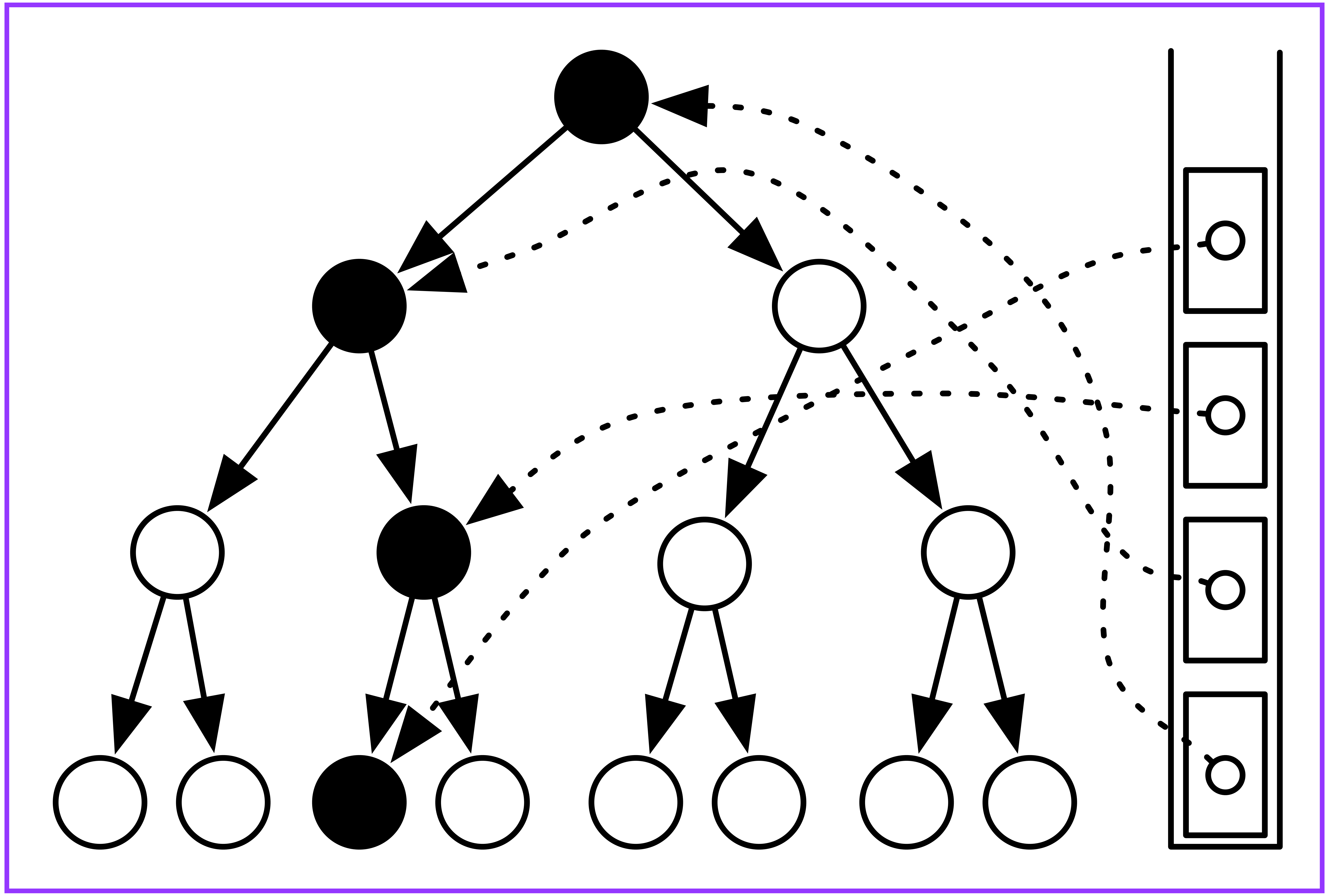
Preorder “display” → prefix notatie

Postorder “display” → postfix notatie

Inorder “display” → infix notatie



Verband Stacks & Boomtraversals



Iteratieve Pre-order Traversal

De stack wordt expliciet

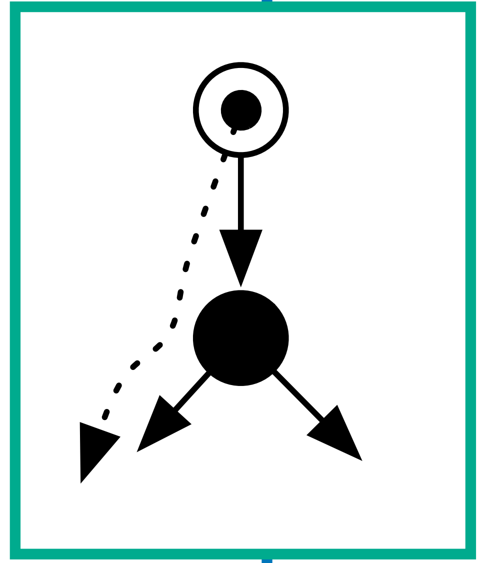
```
(define (iterative-pre-order tree proc)
  (define stack (stack:new))
  (define (loop)
    (if (not (stack:empty? stack))
      (let ((node (stack:pop! stack)))
        (proc (value node))
        (if (not (null-tree? (right node)))
          (stack:push! stack (right node)))
        (if (not (null-tree? (left node)))
          (stack:push! stack (left node)))
        (loop))))
  (stack:push! stack tree)
  (loop))
```

Iteratieve In-order Traversal

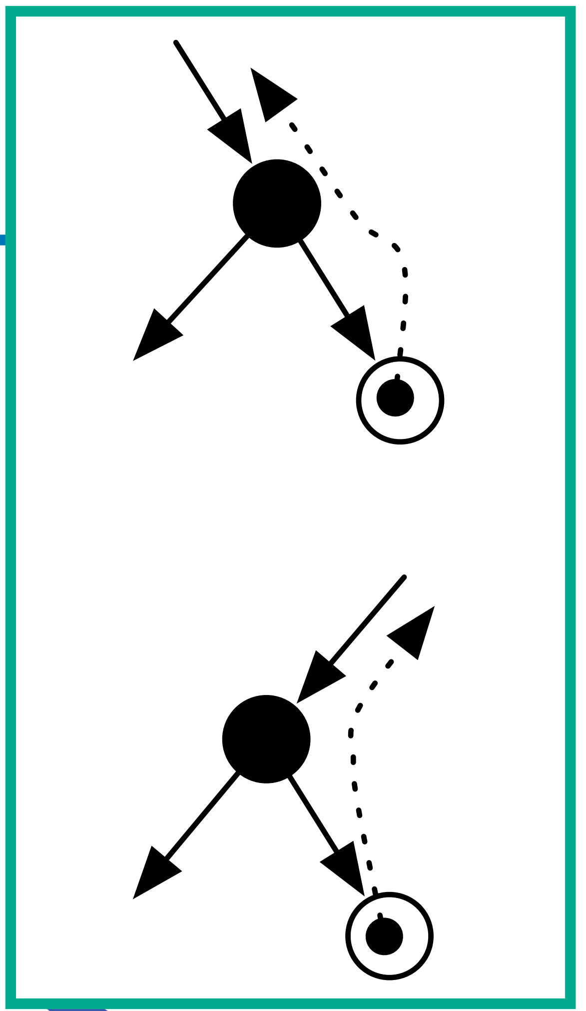
```
(define (iterative-in-order tree proc)
  (define stack (stack:new))
  (define (loop-up)
    (let ((node (stack:pop! stack)))
      (proc (value node))
      (if (not (null-tree? (right node)))
        (begin (stack:push! stack (right node))
               (loop-down))
        (if (not (stack:empty? stack))
          (loop-up)))))
  (define (loop-down)
    (let ((node (stack:top stack)))
      (if (not (null-tree? (left node)))
        (begin (stack:push! stack (left node))
               (loop-down))
        (loop-up))))
  (stack:push! stack tree)
  (loop-down))
```

Iteratieve Post-order Traversal

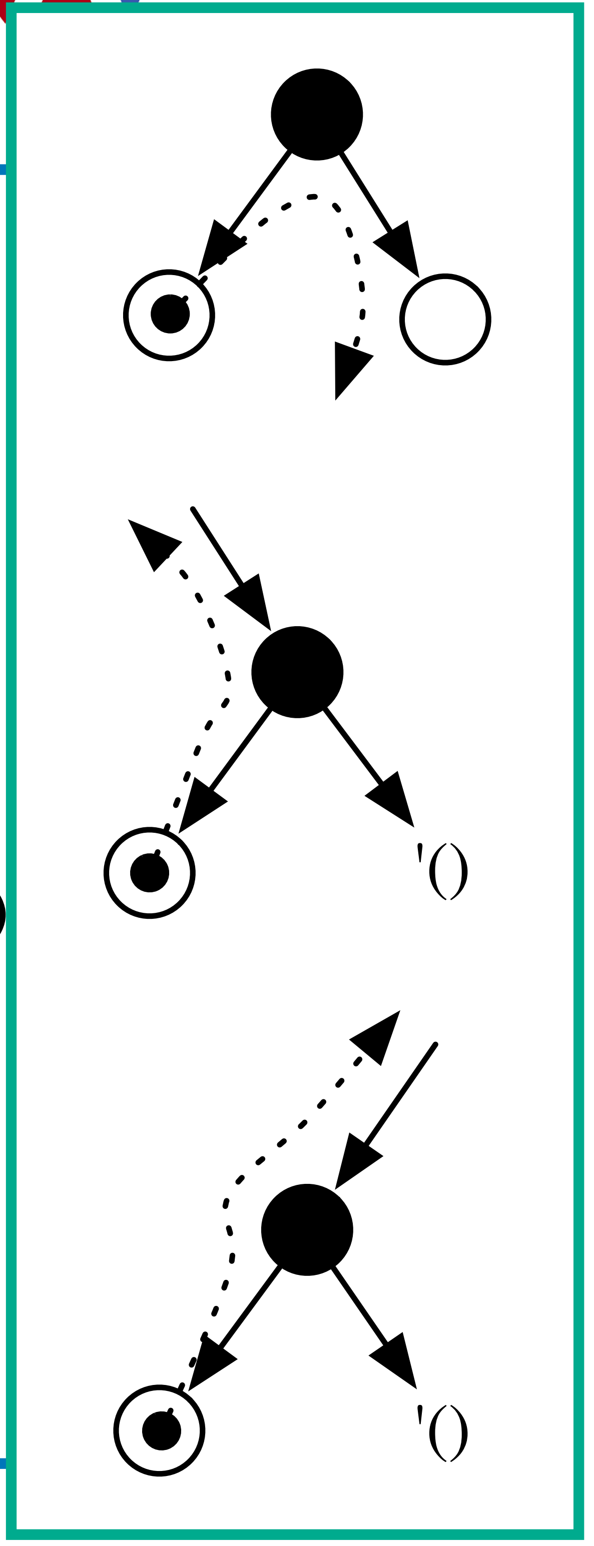
```
(define (iterative-post-order tree proc)
  (define stack (stack:new))
  (define (loop-up-right)
    ...)
  (define (loop-up-left)
    ...)
  (define (loop-down)
    (if (not (stack:empty? stack))
      (let ((node (stack:top stack)))
        (if (null-tree? (left node))
          (loop-up-left)
          (begin
            (stack:push! stack (left node))
            (loop-down))))))
  (stack:push! stack tree)
  (loop-down))
```



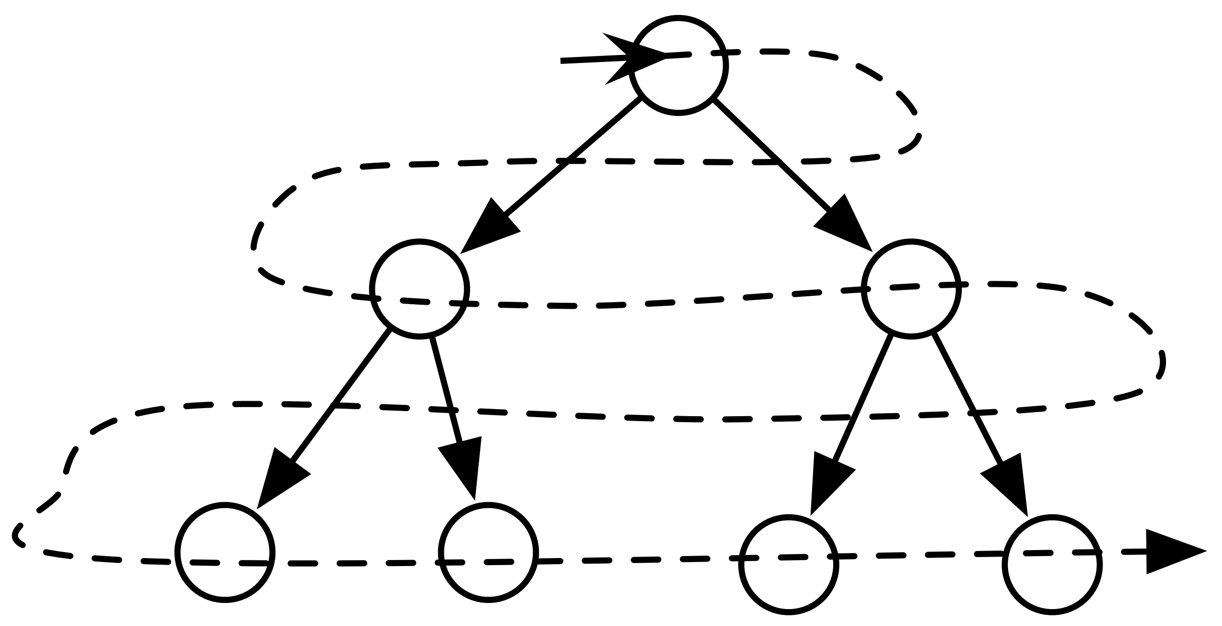
```
(define (loop-up-right)
  (let ((node (stack:pop! stack)))
    (proc (value node))
    (cond ((and (not (stack:empty? stack))
                (eq? (right (stack:top stack)) node))
           (loop-up-right))
          ((not (stack:empty? stack))
           (loop-up-left)))))
```



```
(define (loop-up-left)
  (let ((node (stack:pop! stack)))
    (cond ((not (null-tree? (right node)))
           (stack:push! stack node)
           (stack:push! stack (right node))
           (loop-down))
          ((and (not (stack:empty? stack))
                (eq? (right (stack:top stack)) node))
           (proc (value node))
           (loop-up-right))
          ((not (stack:empty? stack))
           (proc (value node))
           (loop-up-left)))))
```



Breadth-first Traversals



```
(define (breadth-first tree proc)
  (define q (queue:new))
  (define (loop)
    (let ((node (queue:serve! q)))
      (proc (value node))
      (if (not (null-tree? (left node)))
        (queue:enqueue! q (left node)))
      (if (not (null-tree? (right node)))
        (queue:enqueue! q (right node)))
      (if (not (queue:empty? q))
        (loop))))
  (queue:enqueue! q tree)
  (loop))
```

Herinner het dictionary ADT

Key-value paren met keys van type K en values van type V

Associatief geheugen (met elke key wordt een value geassocieerd)

```
ADT dictionary< K V >

new
    ( ( K K → boolean ) → dictionary< K V > )
dictionary?
    ( any → boolean )
insert!
    ( dictionary< K V > K V → dictionary< K V > )
delete!
    ( dictionary< K V > K → dictionary< K V > )
find
    ( dictionary< K V > K → V ∪ {#f} )
empty?
    ( dictionary< K V > → boolean )
full?
    ( dictionary< K V > → boolean )
```

Een dictionary is een datastructuur die "associaties" beheert. Elke associatie bestaat uit een key en een value. De belangrijkste operaties zijn het toevoegen, verwijderen en opzoeken van een associatie. Het snel implementeren van deze 3 operaties is één van de centrale vraagstukken van de cursus.

Implementatie met Sorted Lists

```
ADT sorted-list<V>

new
   ( (V V → boolean)
     (V V → boolean) → sorted-list<V> )
from-scheme-list
   ( pair
     (V V → boolean)
     (V V → boolean) → sorted-list<V>) )
sorted-list?
   ( any → boolean)
length
   ( sorted-list<V> → number )
find!
   ( sorted-list<V> V → sorted-list<V> )
delete!
   ( sorted-list<V> → sorted-list<V> )
peek
   ( sorted-list<V> → V )
add!
   ( sorted-list<V> V → sorted-list<V> )
set-current-to-first!
   ( sorted-list<V> → sorted-list<V> )
set-current-to-next!
```

```
(import (scheme base)
        (prefix slist: (a-d sorted-list linked)))
```

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

```
(define (lift proc)
  (lambda (assoc1 assoc2)
    (proc (assoc-key assoc1)
          (assoc-key assoc2))))

(define (new ==? <<?)
  (slist:new
   (lift <<?)
   (lift ==?)))
```

Performantie

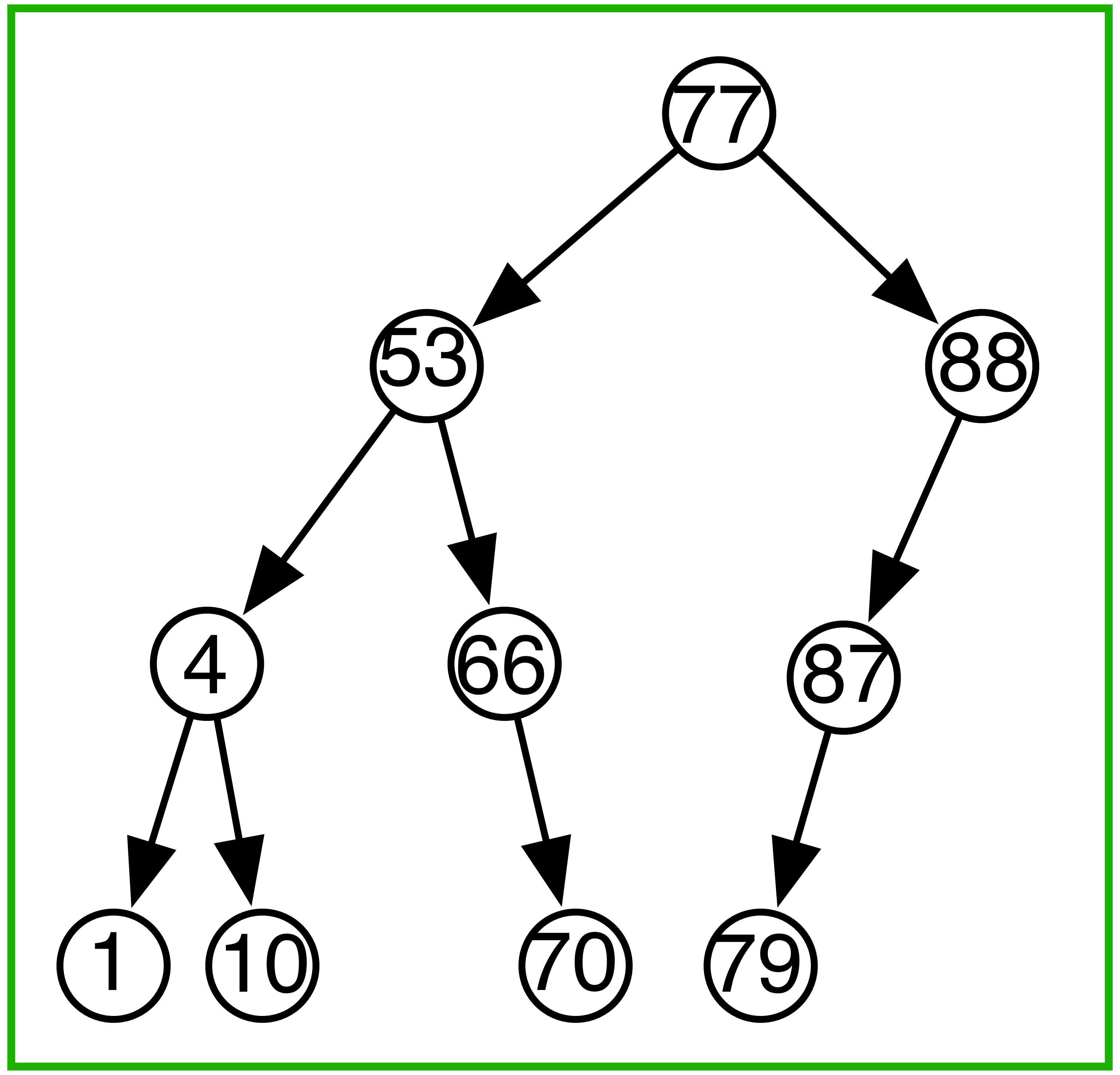
	gelinkte implementatie	vector implementatie	boom implementatie
	sorted list		
new	O(1)	O(1)	O(1)
find	O(n)	O(log(n))	O(log(n))
insert!	O(n)	O(n)	O(log(n))
delete!	O(n)	O(n)	O(log(n))
flexibel	√	∅	√

6.3 Binaire zoekbomen

Binaire Zoekbomen

“De BST Voorwaarde”

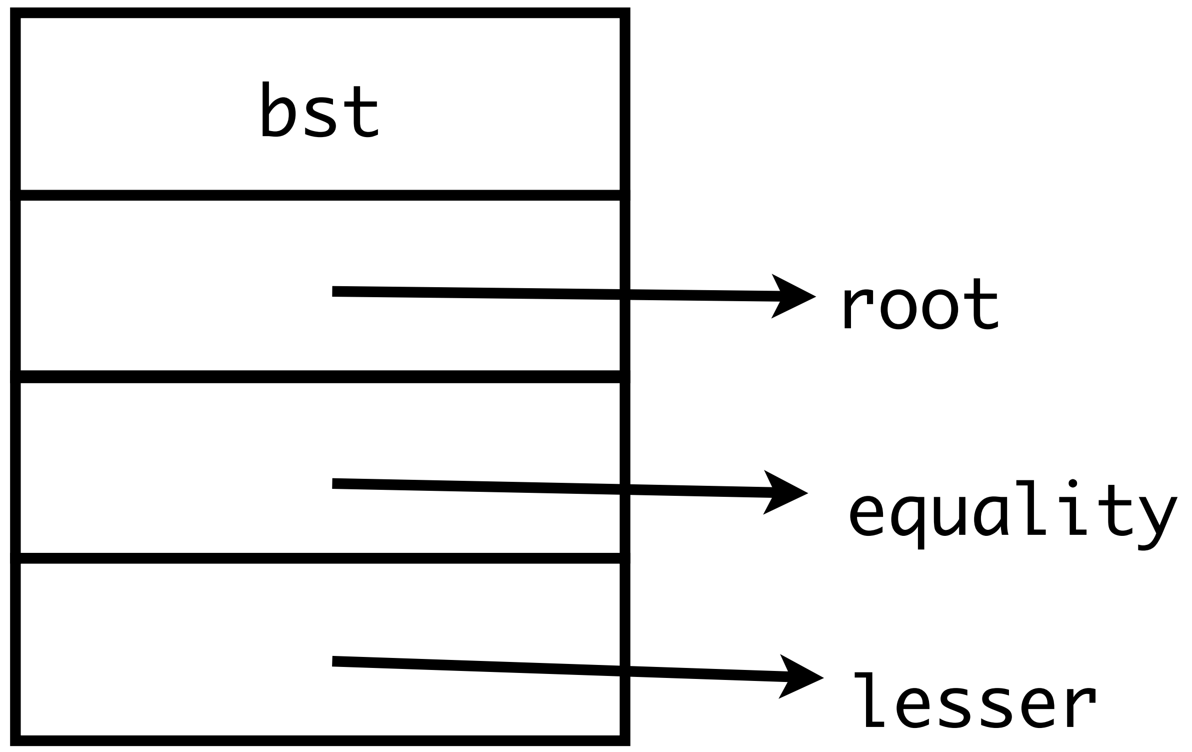
Een binaire zoekboom is een binaire boom zodat voor elke node n geldt: de elementen in de linkerdeelboom van n zijn allen kleiner dan n en de elementen in de rechterdeelboom van n zijn allen groter dan n.



Binaire Zoekbomen: ADT

```
ADT BST<V>

new
   ( ( V V → boolean) ( V V → boolean) → BST<V> )
tree?
   ( any → boolean )
find
   ( BST<V> V → V ∪ { #f } )
insert!
   ( BST<V> V → BST<V> )
delete!
   ( BST<V> V → BST<V> )
```

Representatie




```
(define-record-type bst
  (make r e l)
  bst?
  (r root root!)
  (e equality)
  (l lesser))

(define (new ==? <<?)
  (make tree:null-tree ==? <<?))
```

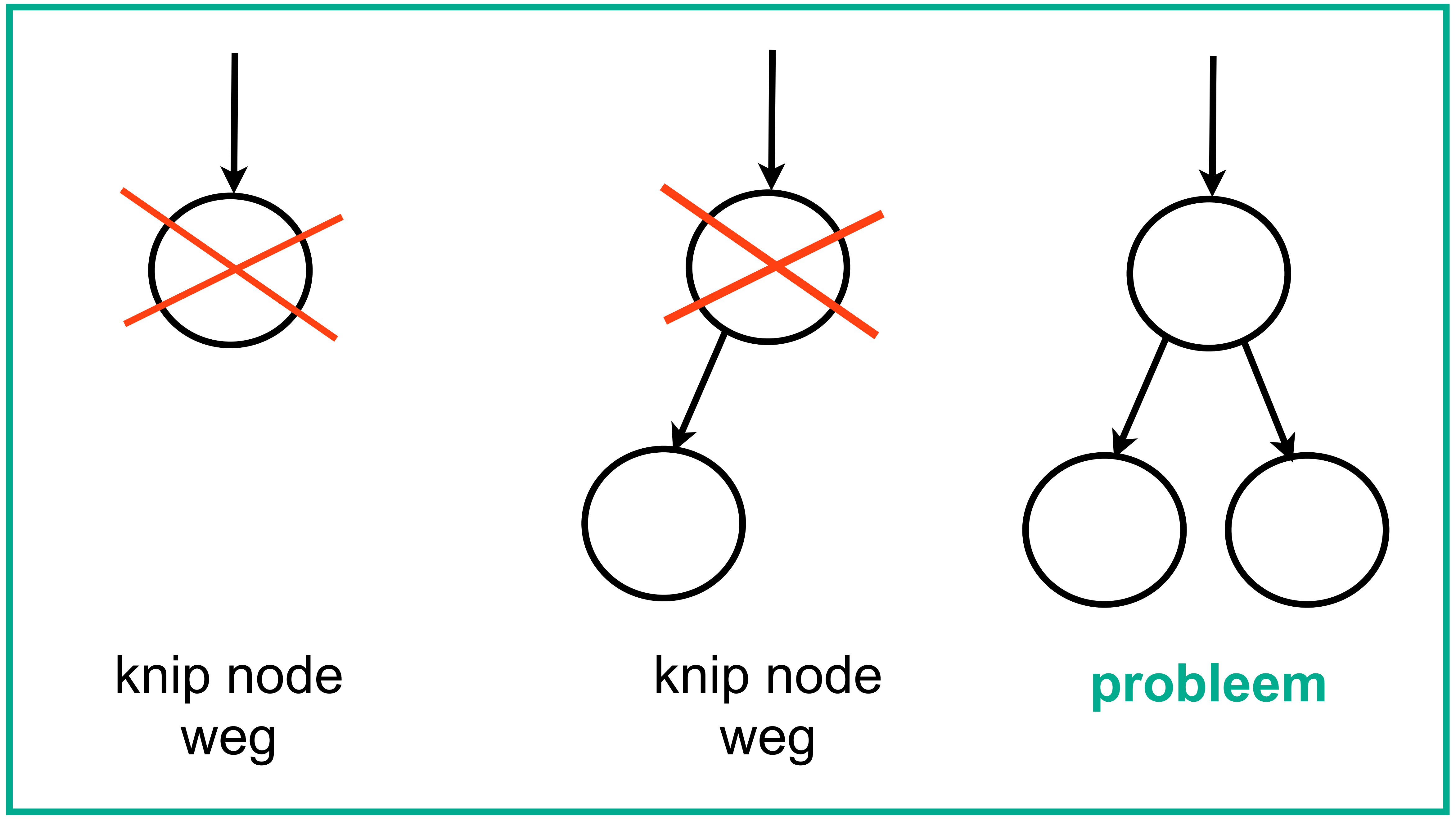
Implementatie

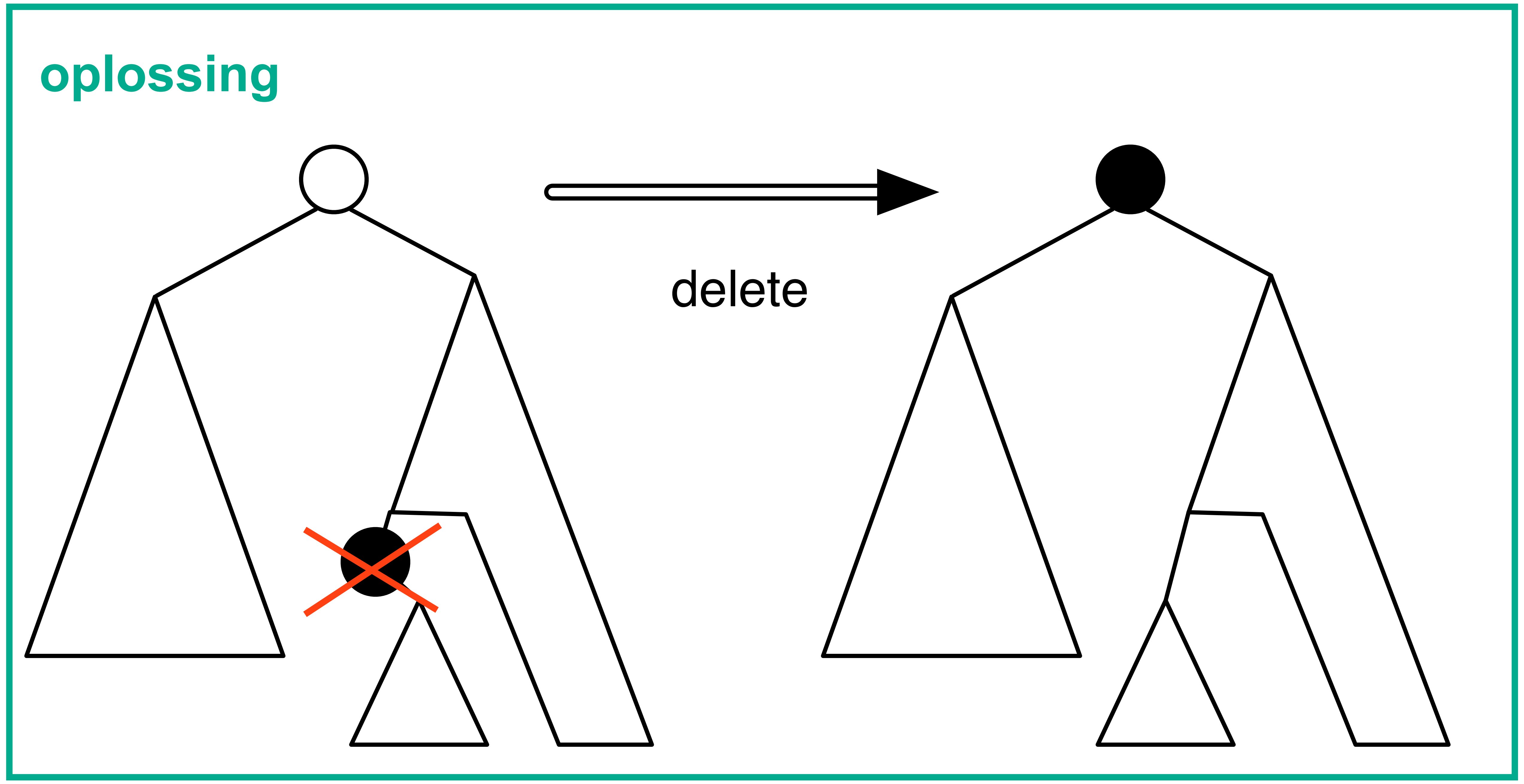
```
(define (find bst key)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (let find-key
    ((node (root bst)))
    (if (tree:null-tree? node)
      #f
      (let
          ((node-value (tree:value node)))
        (cond
          ((==? node-value key)
           node-value)
          ((<<? node-value key)
           (find-key (tree:right node)))
          ((<<? key node-value)
           (find-key (tree:left node))))))))
```

```
(define (insert! bst val)
  (define <<? (lesser bst))
  (let insert-iter
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child  (root bst)))
    (cond
      ((tree:null-tree? child)
       (child! parent
               (tree:new val
                         tree:null-tree
                         tree:null-tree)))
      ((<<? (tree:value child) val)
       (insert-iter child tree:right!
                    (tree:right child)))
      ((<<? val (tree:value child))
       (insert-iter child tree:left!
                    (tree:left child)))
      (else
       (tree:value! child val)))))
```

insert! is zoals find. plak het element waar je het verwacht terug te vinden.

Deleten uit een BST





Implementatie

```
(define (delete! bst val)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (define (find-leftmost deleted parent child! child)
    ...)
  (define (delete-node parent child! child)
    ...)

  (let find-node
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! bst child)))
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       #f)
      ((==? (tree:value child) val)
       (delete-node parent child! child)
       (tree:value child))
      ((<<? (tree:value child) val)
       (find-node child tree:right! (tree:right child)))
      ((<<? val (tree:value child))
       (find-node child tree:left! (tree:left child))))))
```

```
(define (find-leftmost deleted parent child! child)
  (if (tree:null-tree? (tree:left child))
    (begin
      (tree:value! deleted (tree:value child))
      (child! parent (tree:right child)))
    (find-leftmost deleted child
                   tree:left!
                   (tree:left child))))
```

```
(define (delete-node parent child! child)
   (cond
     ((tree:null-tree? (tree:left child))
      (child! parent (tree:right child)))
     ((tree:null-tree? (tree:right child))
      (child! parent (tree:left child)))
     (else
      (find-leftmost child
                     child
                     tree:right!
                     (tree:right child)))))
```

Herinner het dictionary ADT

```
ADT dictionary< K V >

new
    ( ( K K → boolean ) → dictionary< K V > )
dictionary?
    ( any → boolean )
insert!
    ( dictionary< K V > K V → dictionary< K V > )
delete!
    ( dictionary< K V > K → dictionary< K V > )
find
    ( dictionary< K V > K → V ∪ {#f} )
empty?
    ( dictionary< K V > → boolean )
full?
    ( dictionary< K V > → boolean )
```

Key-value paren met keys van type K en values van type V

Associatief geheugen (met elke key wordt een value geassocieerd)

Een dictionary is een datastructuur die "associaties" beheert. Elke associatie bestaat uit een key en een value. De belangrijkste operaties zijn het toevoegen, verwijderen en opzoeken van een associatie. Het snel implementeren van deze 3 operaties is één van de centrale vraagstukken van de cursus.

Implementatie met BSTs

```
(define (insert! dct key val)
  (bst:insert! dct (make-assoc key val))
  dct)

(define (delete! dct key)
  (bst:delete! dct (make-assoc  key 'ignored))
  dct)

(define (find dct key)
  (define assoc (bst:find dct (make-assoc key 'ignored)))
  (if assoc
    (assoc-value assoc)
    assoc))

(define (empty? dct)
  (bst:empty? dct))

(define (full? dct)
  (bst:full? dct))
```

```
(define make-assoc cons)
(define assoc-key car)
(define assoc-value cdr)
```

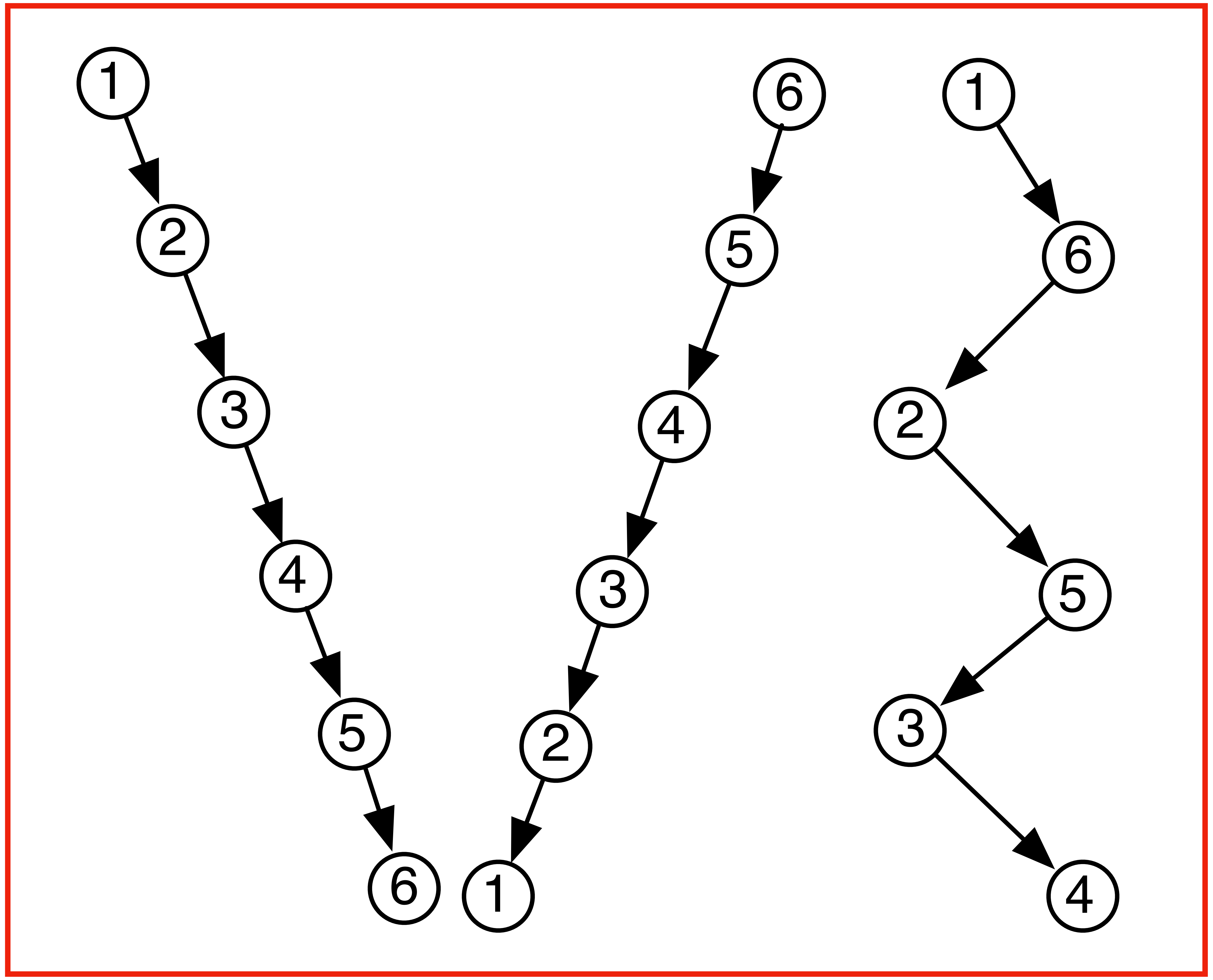
Dictionaries: Performantie

	gelinkte implementatie (sorted list)	vector implementatie (sorted list)	BST implementatie (best case)	BST implementatie (worst case)
new	O(1)	O(1)	O(1)	O(1)
find	O(n)	O(log(n))	Ω(log(n))	O(n)
insert!	O(n)	O(n)	Ω(log(n))	O(n)
delete!	O(n)	O(n)	Ω(log(n))	O(n)

average tree: 1,39log(n)

6.4 AVL bomen

Gedegenereerde Bomen



Erger dan gelinkte lijsten

Er bestaan

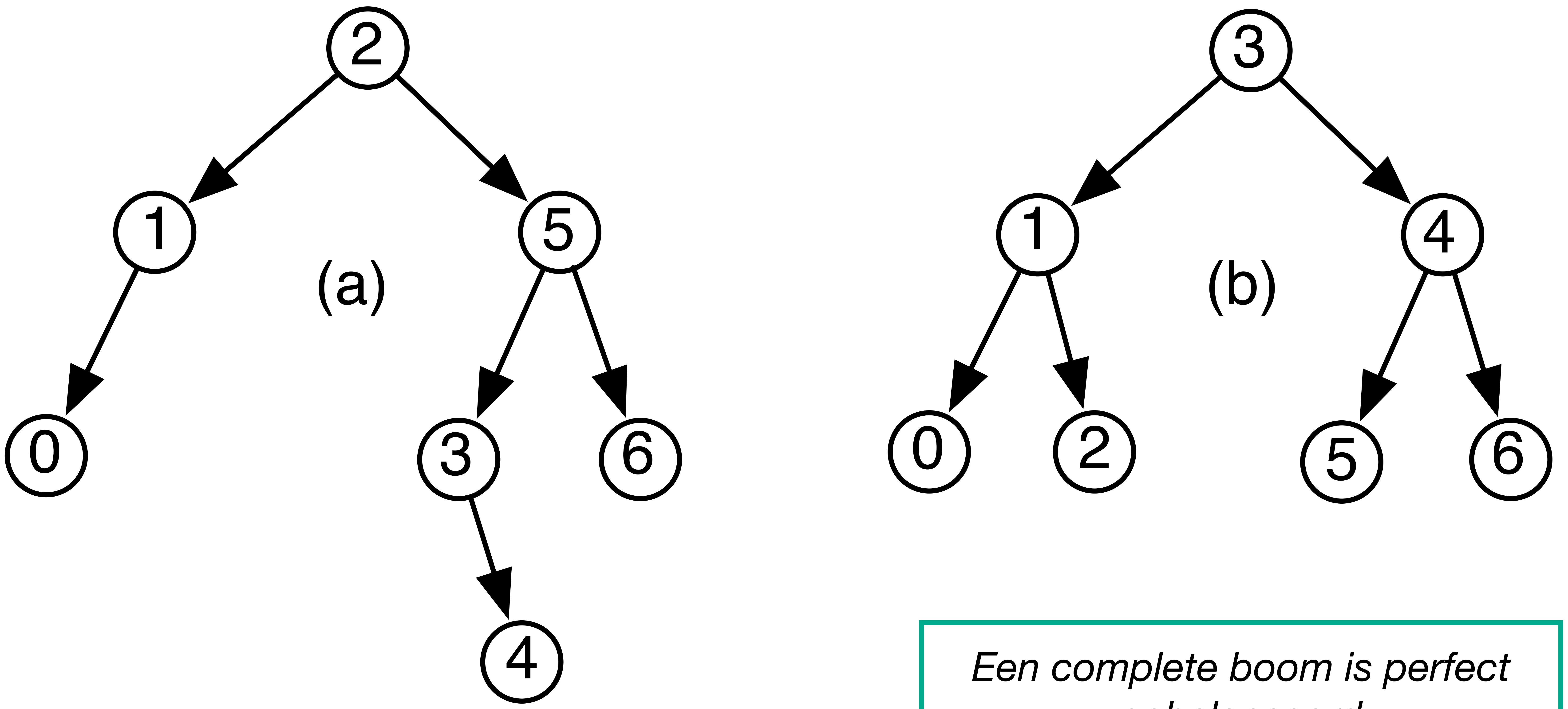
$$\frac{1}{n+1}\binom{2n}{n}$$

zoekbomen met n knopen.

Er bestaan 2^{n-1} gedegenereerde bomen met n knopen

veel!

Antwoord: Gebalanceerde Bomen



Een complete boom is perfect gebalanceerd

Perfect balanceren na elke insert&delete is duur

Zwakkere vormen van Balans

AVL-Bomen

Splaybomen

Rood/Zwart-Bomen

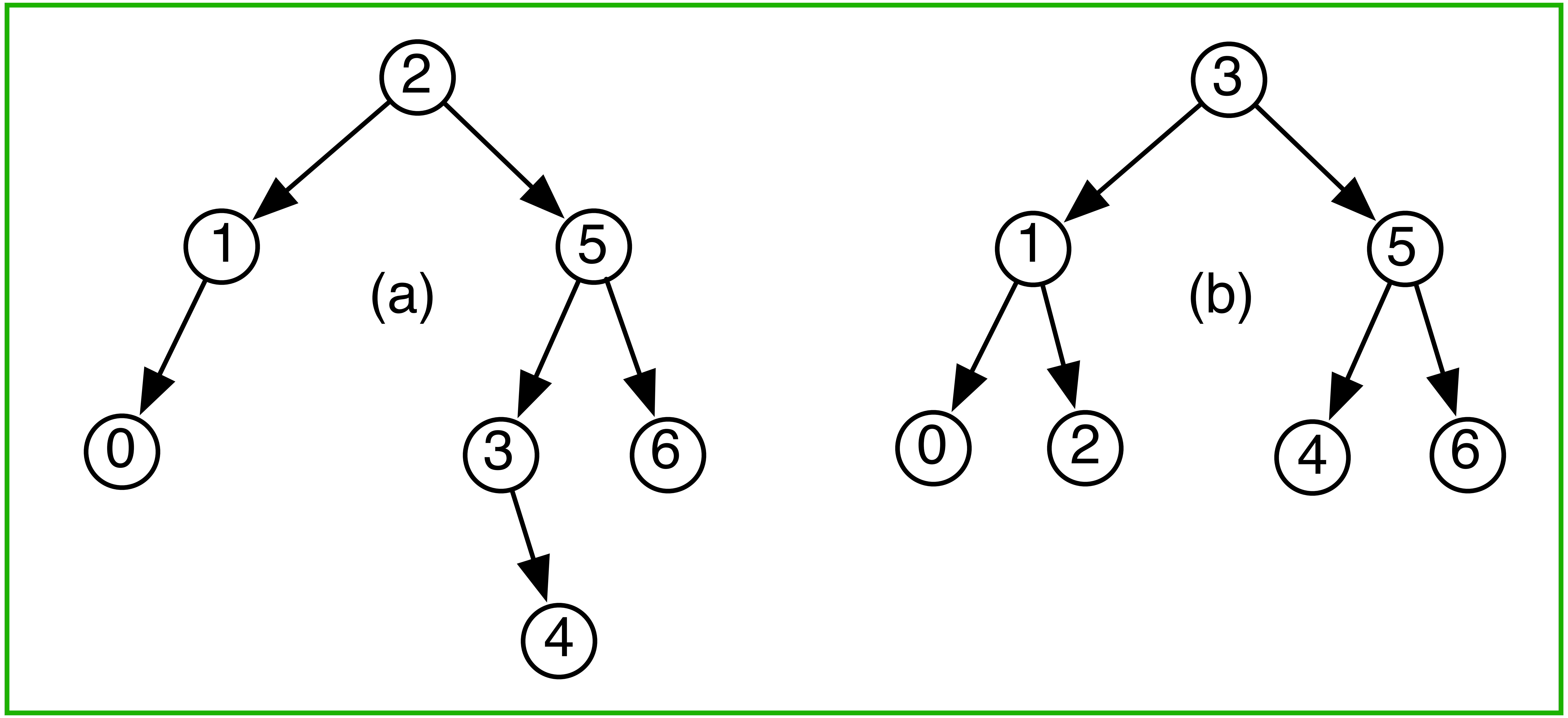
2,3-Bomen

Een AVL-boom is een zoekboom waarbij het hoogteverschil tussen beide subbomen van elke node maximaal 1 is.

Adelson-Velskii & Landis (1962)

$$\lfloor \log_2(n) \rfloor \leq h \leq 1.44 \log_2(n)$$

Voorbeelden



Iedere node is 'balanced, 'Rhigh *of* 'Lhigh

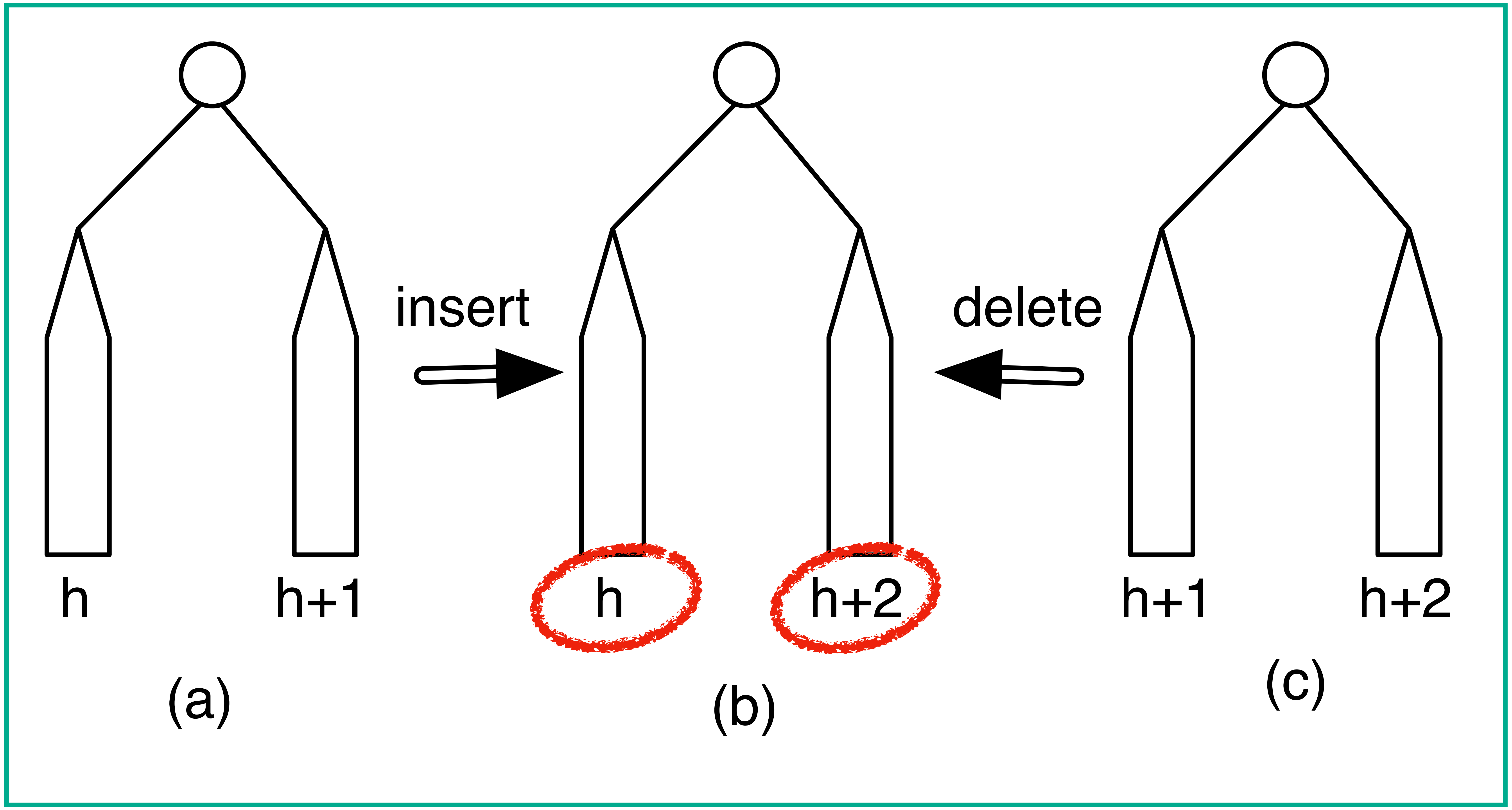
Representatie AVL Bomen

```
(define balanced 'balanced)
(define Lhigh 'Lhigh)
(define Rhigh 'Rhigh)
```

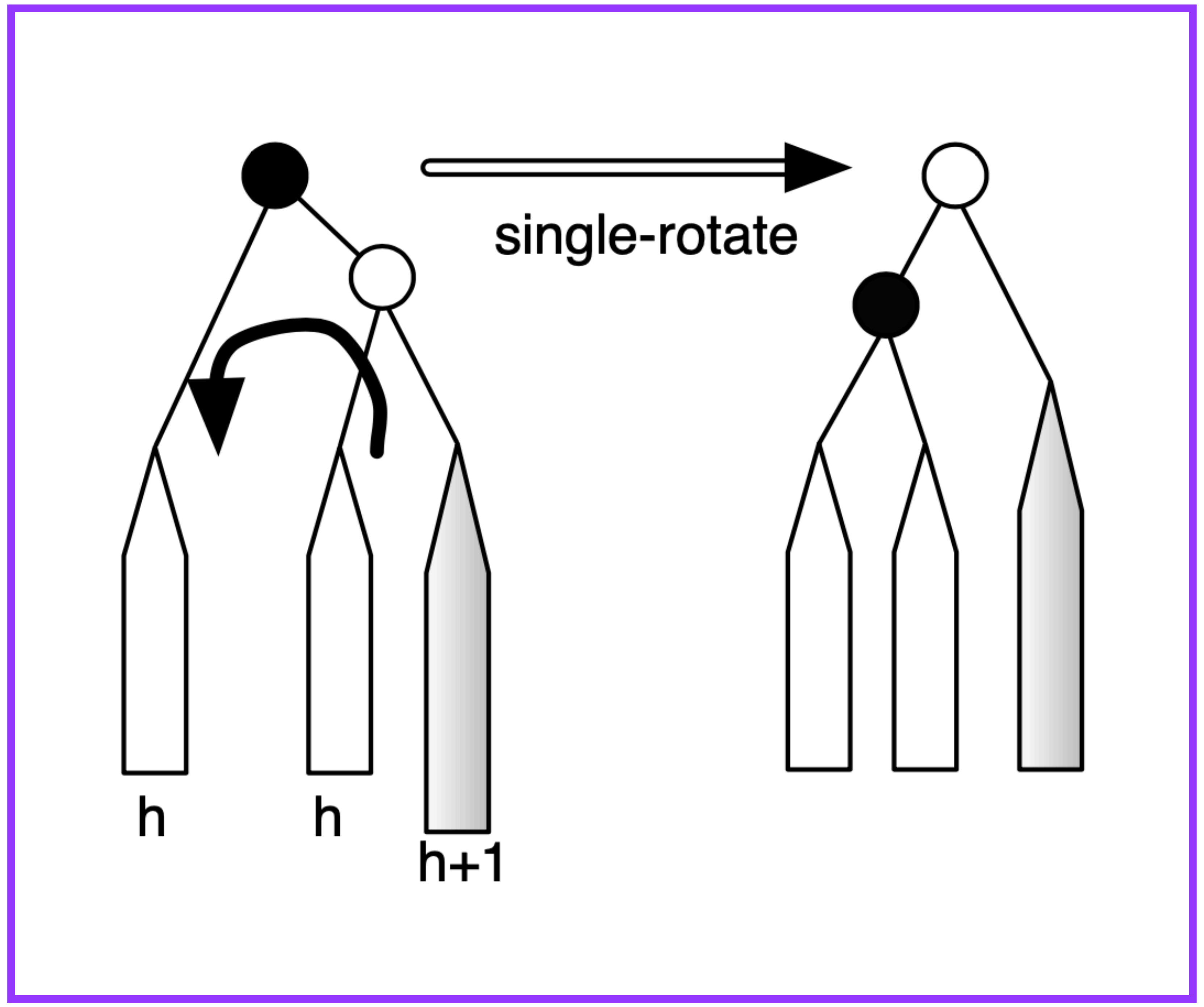
```
(define-record-type AVL-node
  (make-AVL-node v l r b)
  AVL-node?
  (v value value!)
  (l left left!)
  (r right right!)
  (b balance balance!))

(define null-tree ())

(define (null-tree? node)
  (eq? node null-tree))
```

```
(define-record-type bst
  (make r e l)
  bst?
  (r root root!)
  (e equality)
  (l lesser))

 (define (new ==? <<?)
  (make null-tree ==? <<?))
```

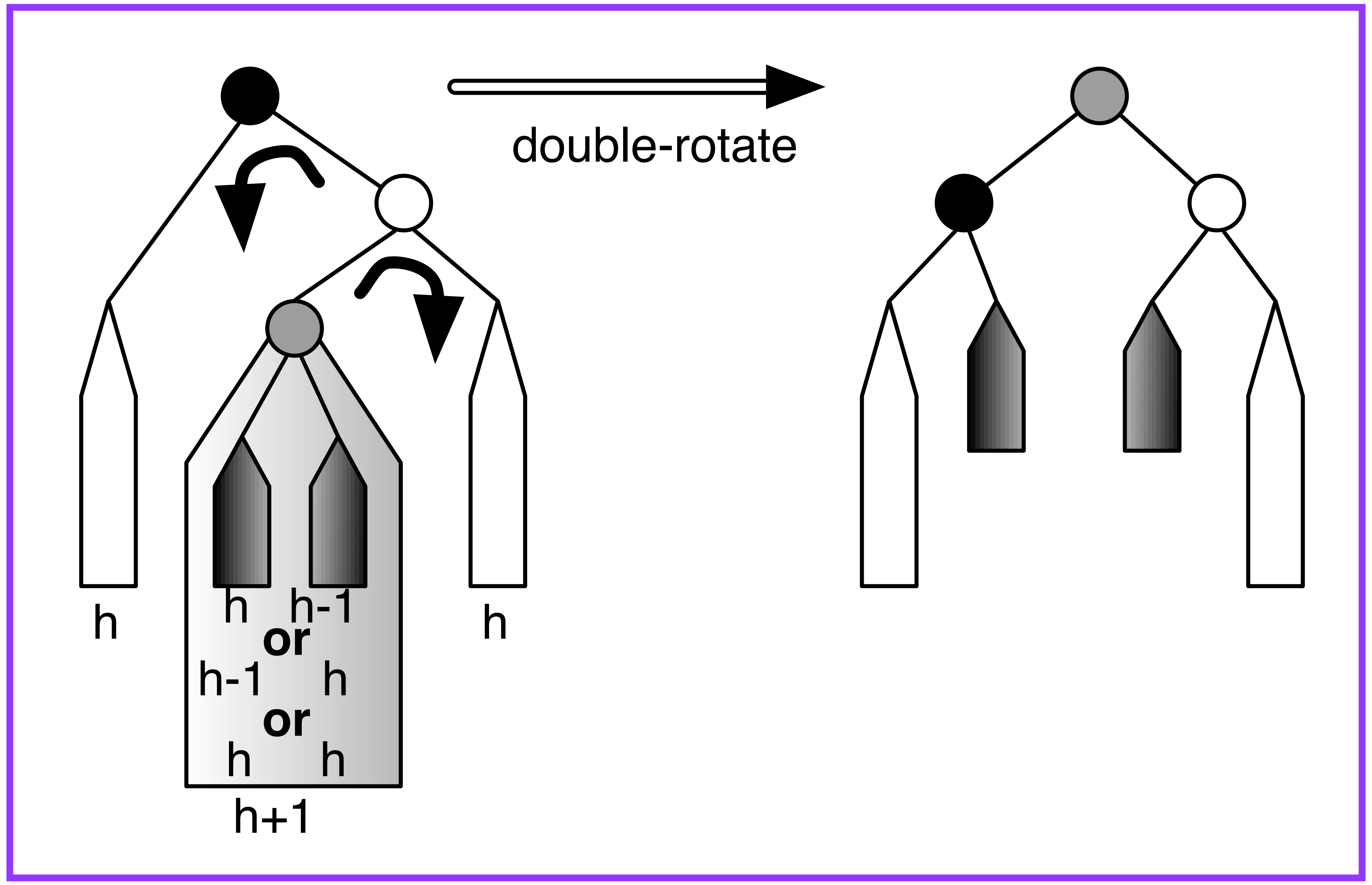
Wat kan er mislopen?



Oplossing: Herbalanceren door Roteren

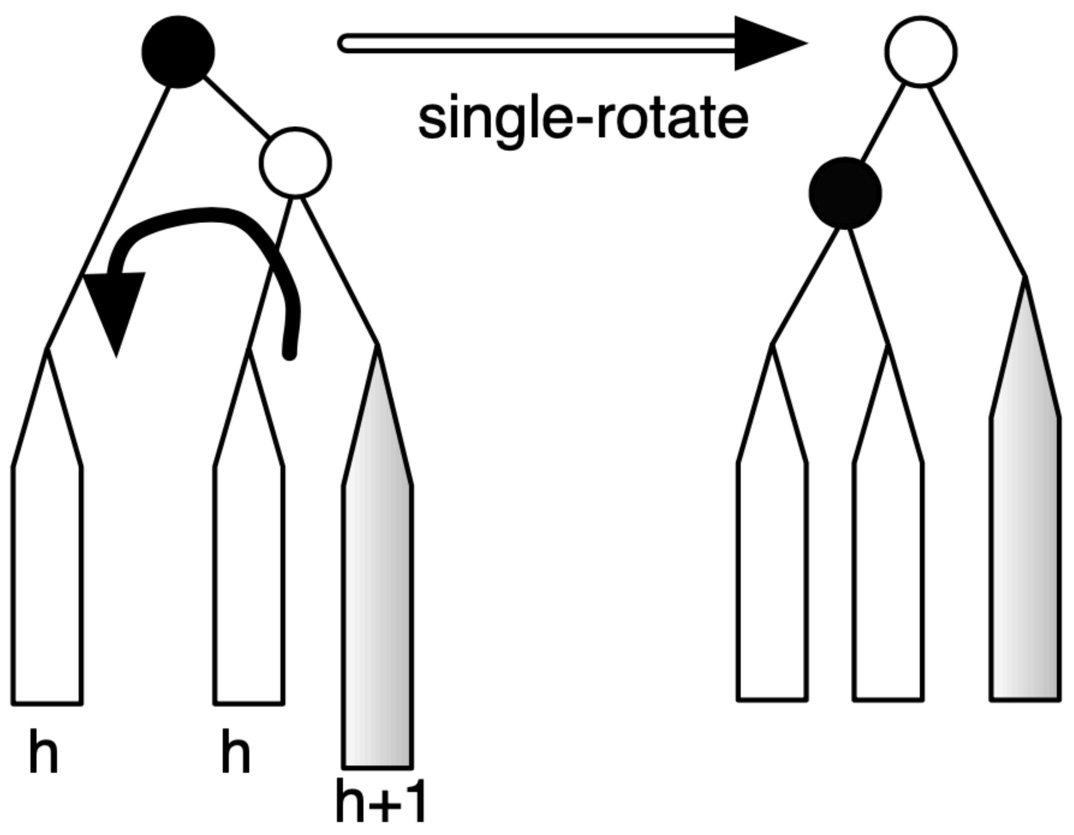


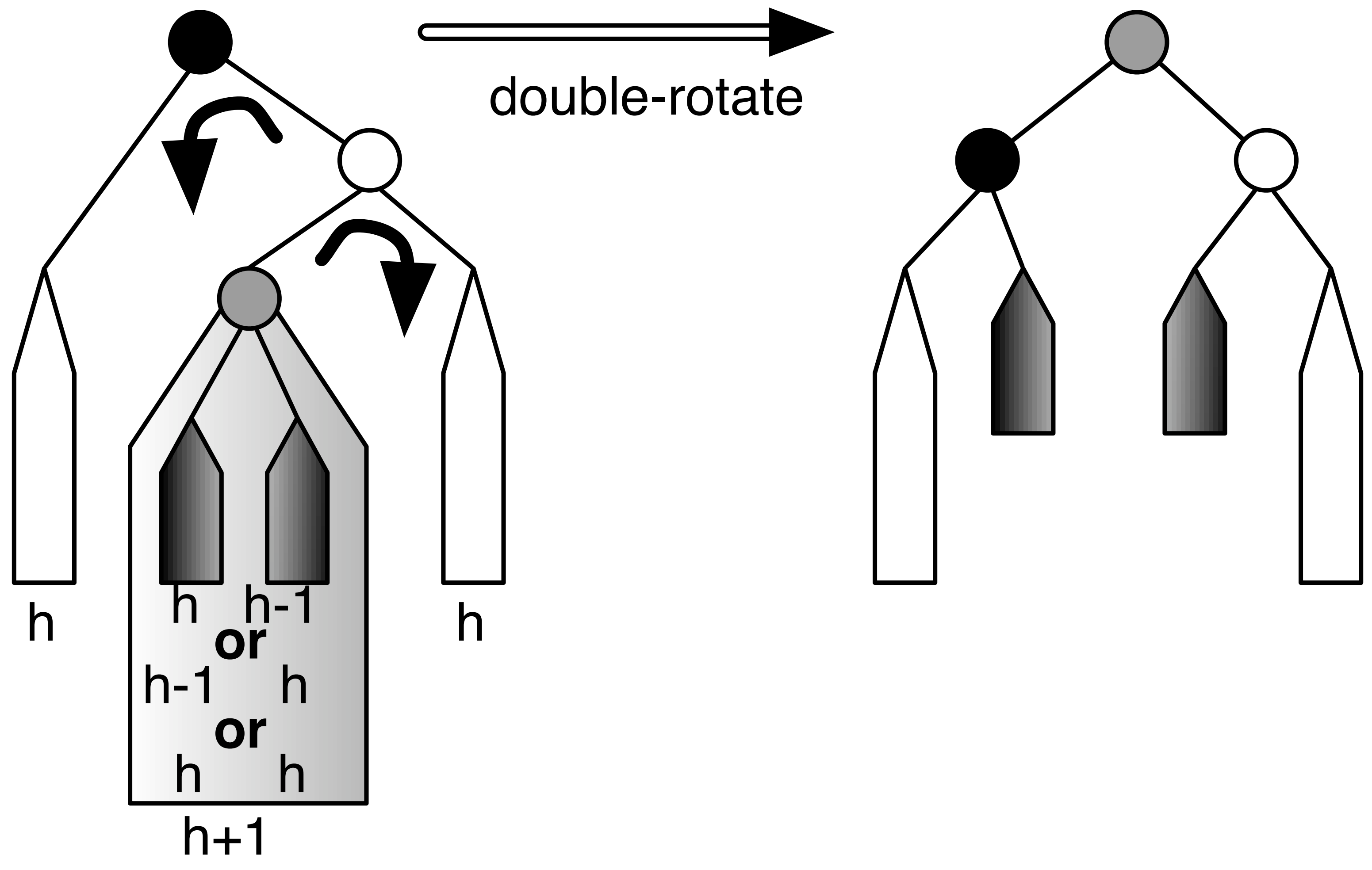
`single-rotate-left!`



`double-rotate-right-then-left!`

4 Rotaties

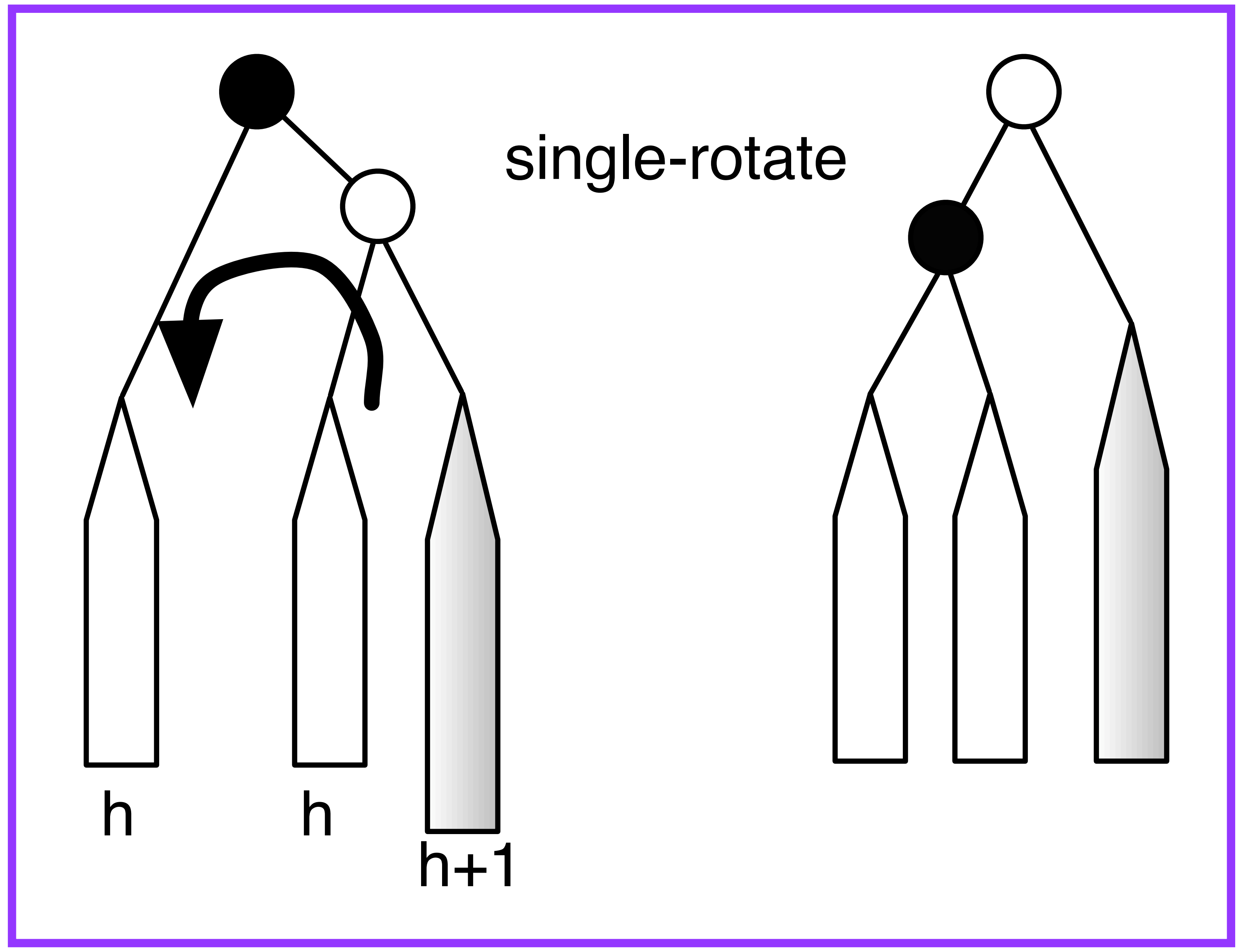




O(1)

```
(define (single-rotate-left! black)
  (define white (right black))
  (define tree (left white))
  (right! black tree)
  (left! white black)
  white)

(define (single-rotate-right! black)
  (define white (left black))
  (define tree (right white))
  (left! black tree)
  (right! white black)
  white)

(define (double-rotate-left-then-right! black)
  (define white (left black))
  (left! black (single-rotate-left! white))
  (single-rotate-right! black))

(define (double-rotate-right-then-left! black)
  (define white (right black))
  (right! black (single-rotate-right! white))
  (single-rotate-left! black))
```

Balanceringsinformatie Updaten (1/2)



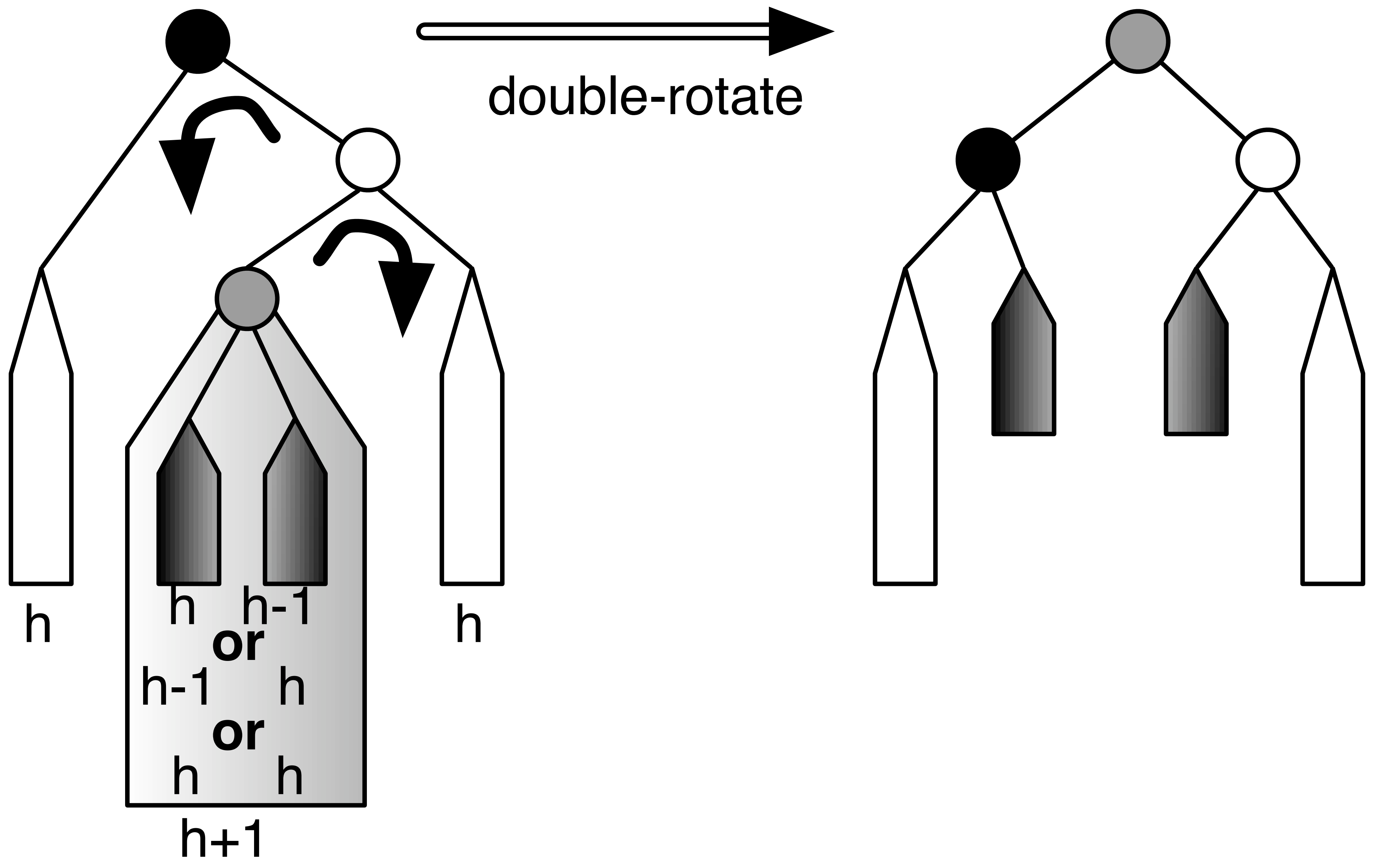
Witte node: balanced

Zwarte node: balanced

Enkel nodig bij delete

```
(define (single-rotate-left-update! black white)
  (cond ((eq? (balance white) Rhigh)
         (balance! black balanced)
         (balance! white balanced))
        (else
         (balance! black Rhigh)
         (balance! white Lhigh))))
```

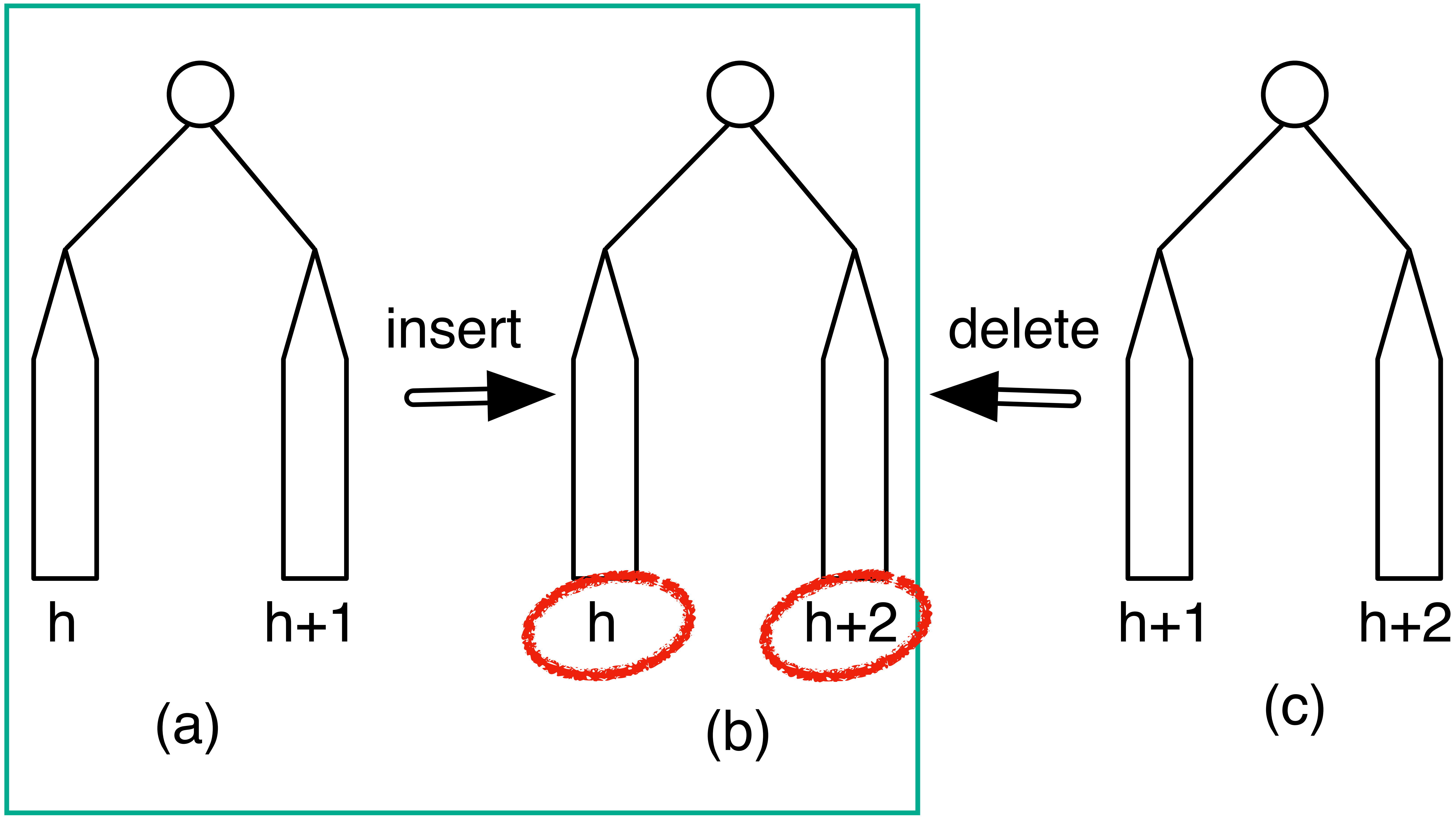
Balanceringsinformatie Updaten (2/2)



Status Grijs:	Lhigh	balanced	Rhigh
Witte node:	Rhigh	balanced	balanced
Zwarte node:	balanced	balanced	Lhigh
Grijze node:	balanced	balanced	balanced

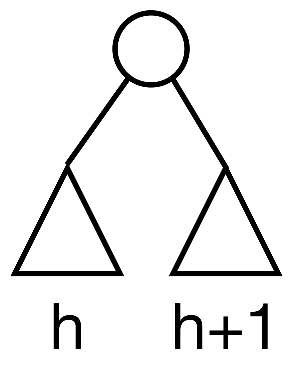
```
(define (double-rotate-right-then-left-update! black white grey)
  (cond ((eq? (AVL-node-balance grey) Lhigh)
          (AVL-node-balance! white Rhigh)
          (AVL-node-balance! black balanced)
          (AVL-node-balance! grey balanced))
        ((eq? (AVL-node-balance grey) balanced)
          (AVL-node-balance! white balanced)
          (AVL-node-balance! black balanced)
          (AVL-node-balance! grey balanced))
        (else
          (AVL-node-balance! white balanced)
          (AVL-node-balance! black Lhigh)
          (AVL-node-balance! grey balanced))))
```

Wat kan er mislopen na insert?



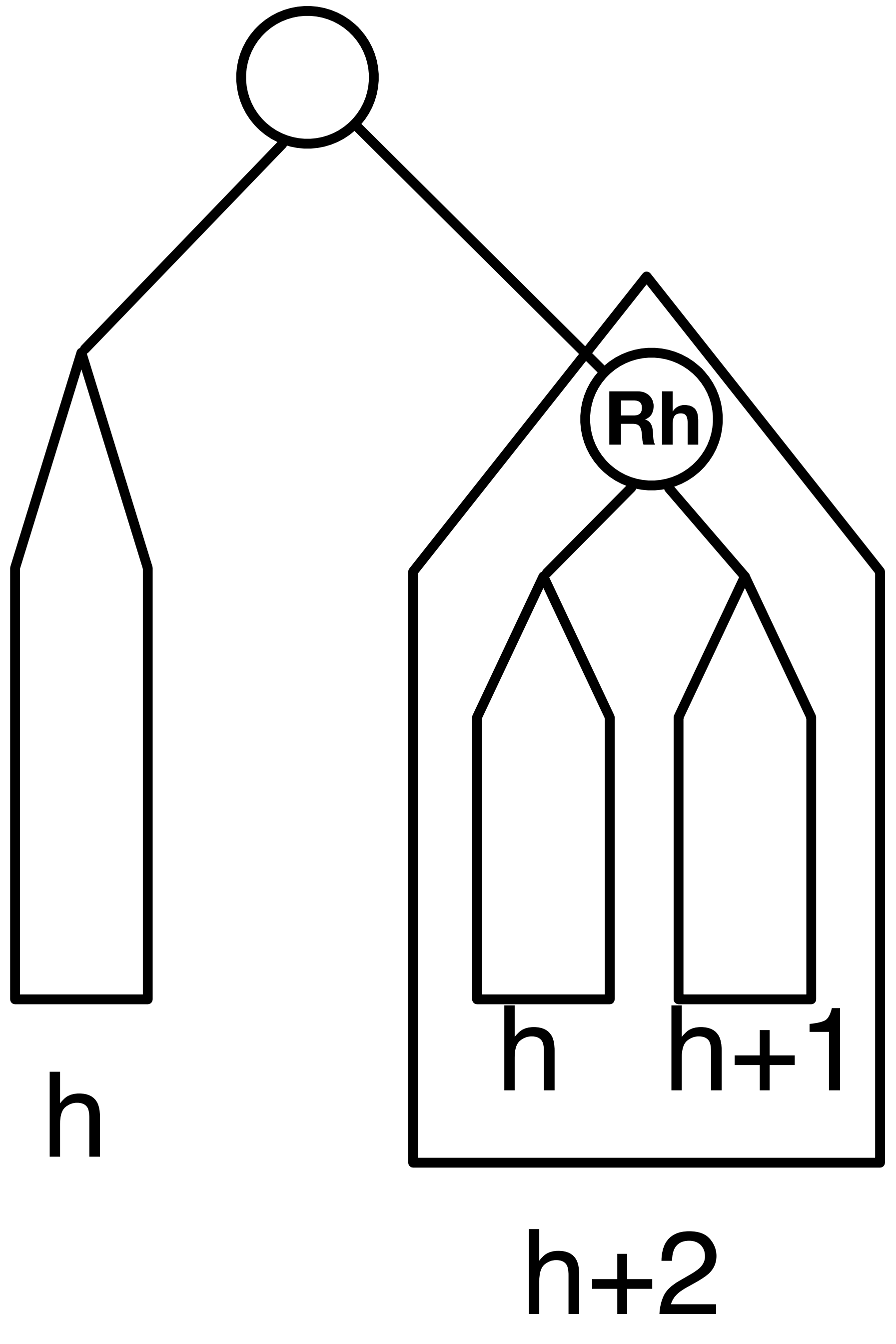
We concentreren ons nu op geval (b) na (a)

Geval (b) na (a): in detail

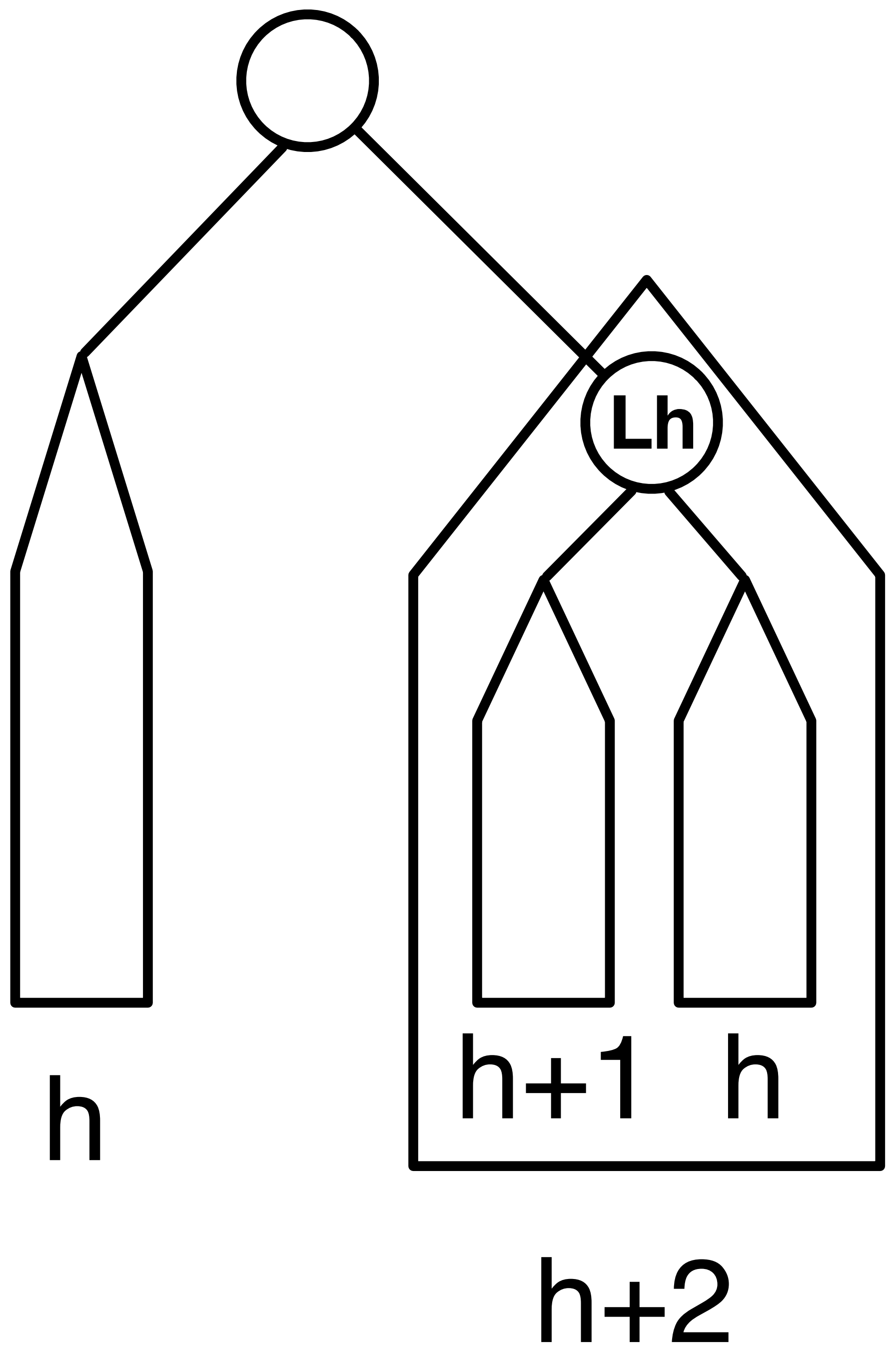


(a)

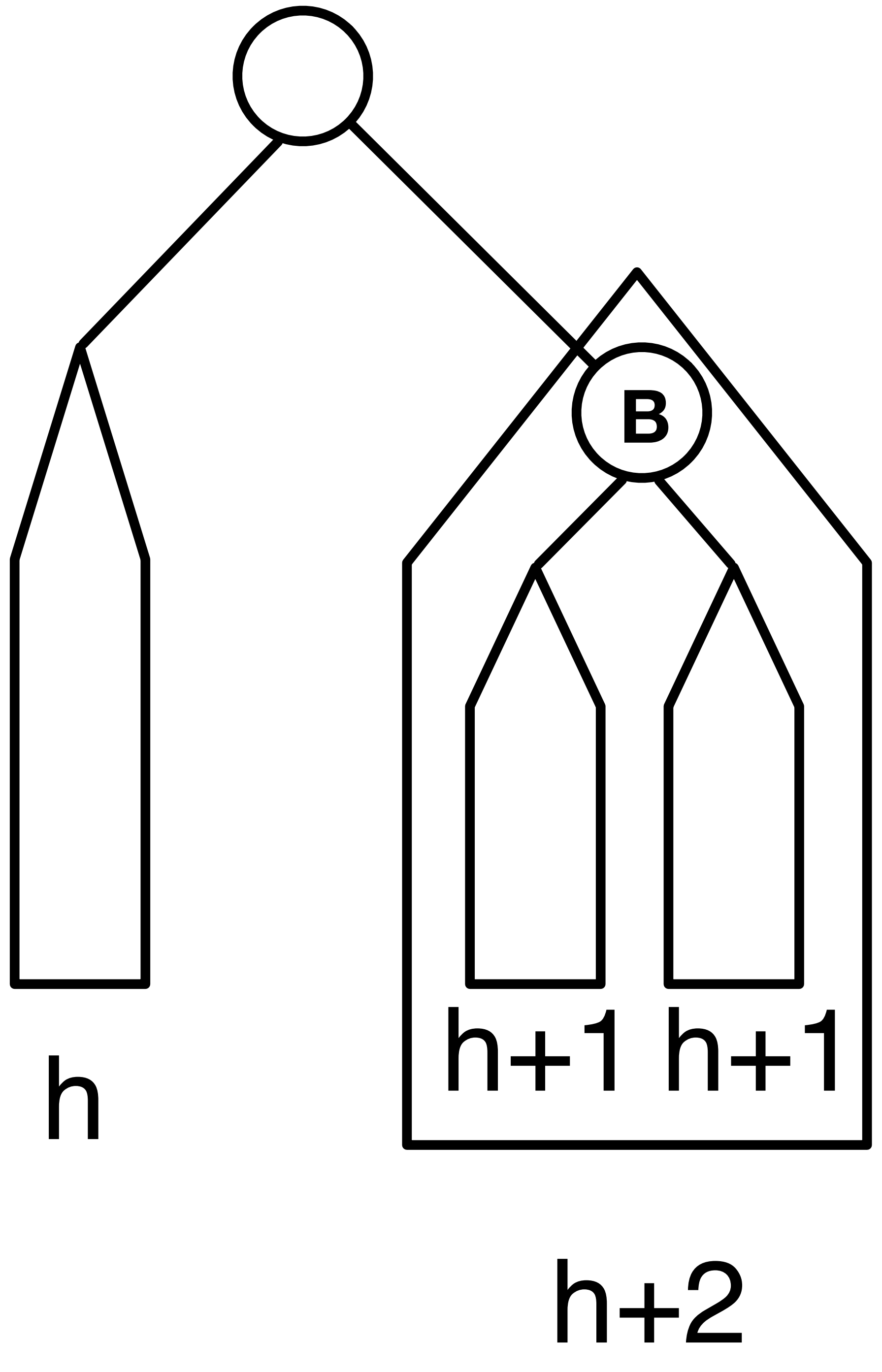
⇒



Apply Single Rotation

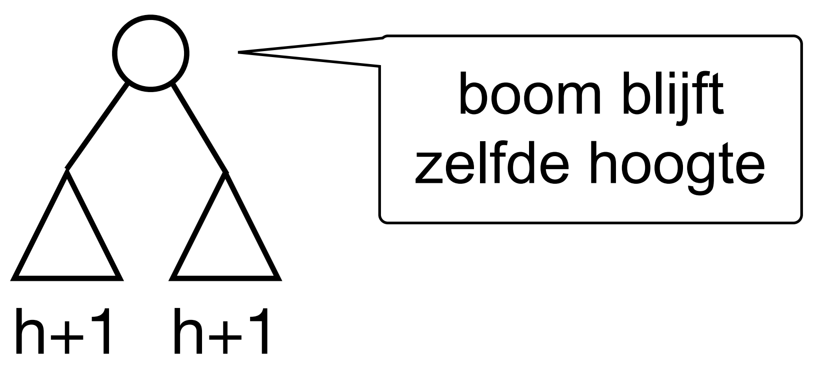


Apply Double Rotation

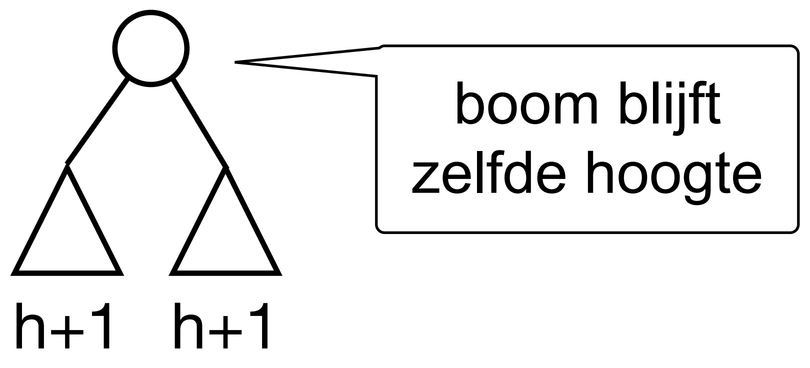


Impossible Situation

⇒



⇒



Het insert algoritme

```
(define (insert! bst val)
  (define <<? (lesser bst))
  (let insert-iter
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root
     (child  (root bst)))
    (cond
      ((tree:null-tree? child)
       (child! parent
               (tree:new val
                         tree:null-tree
                         tree:null-tree)))
      ((<<? (tree:value child) val)
       (insert-iter child tree:right!
                    (tree:right child)))
      ((<<? val (tree:value child))
       (insert-iter child tree:left!
                    (tree:left child)))
      (else
       (tree:value! child val)))))
```

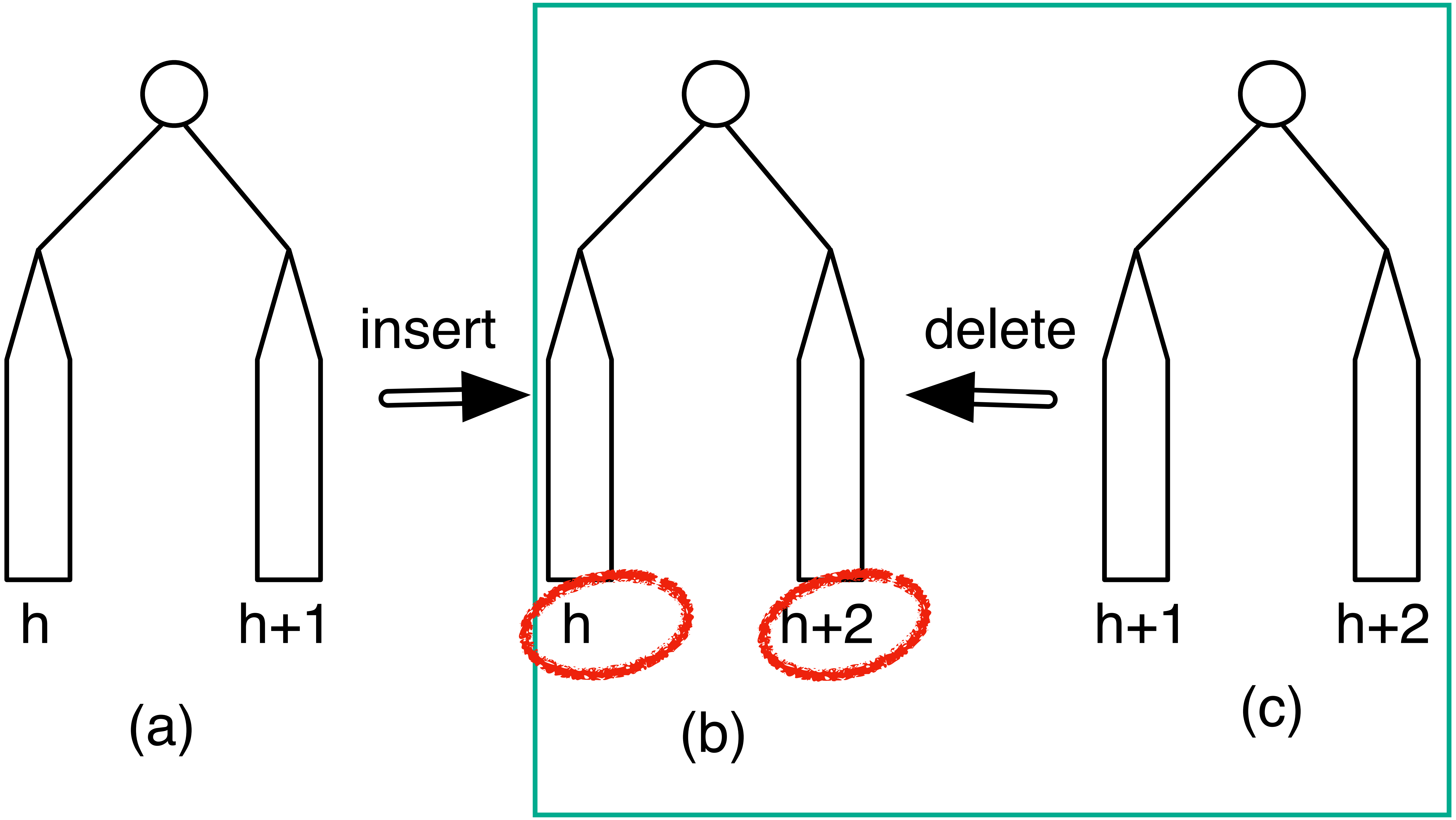
```
(define (insert! avl val)
  (define <<? (lesser avl))
  (define ==? (equality avl))

  (let insert-rec
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       (child! parent (make-AVL-node null-tree val balanced null-tree))
       #t)
      ((<<? (AVL-node-value child) val)
       (if (insert-rec child AVL-node-right! (AVL-node-right child))
         (check-after-insert-right parent child! child)
         #f))
      ((<<? val (AVL-node-value child))
       (if (insert-rec child AVL-node-left! (AVL-node-left child))
         (check-after-insert-left parent child! child)
         #f))
      (else
       (AVL-node-value! child val)
       #f)))
  avl)
```

Eventueel herbalanceren na rechts inserten

Het andere geval is symmetrisch

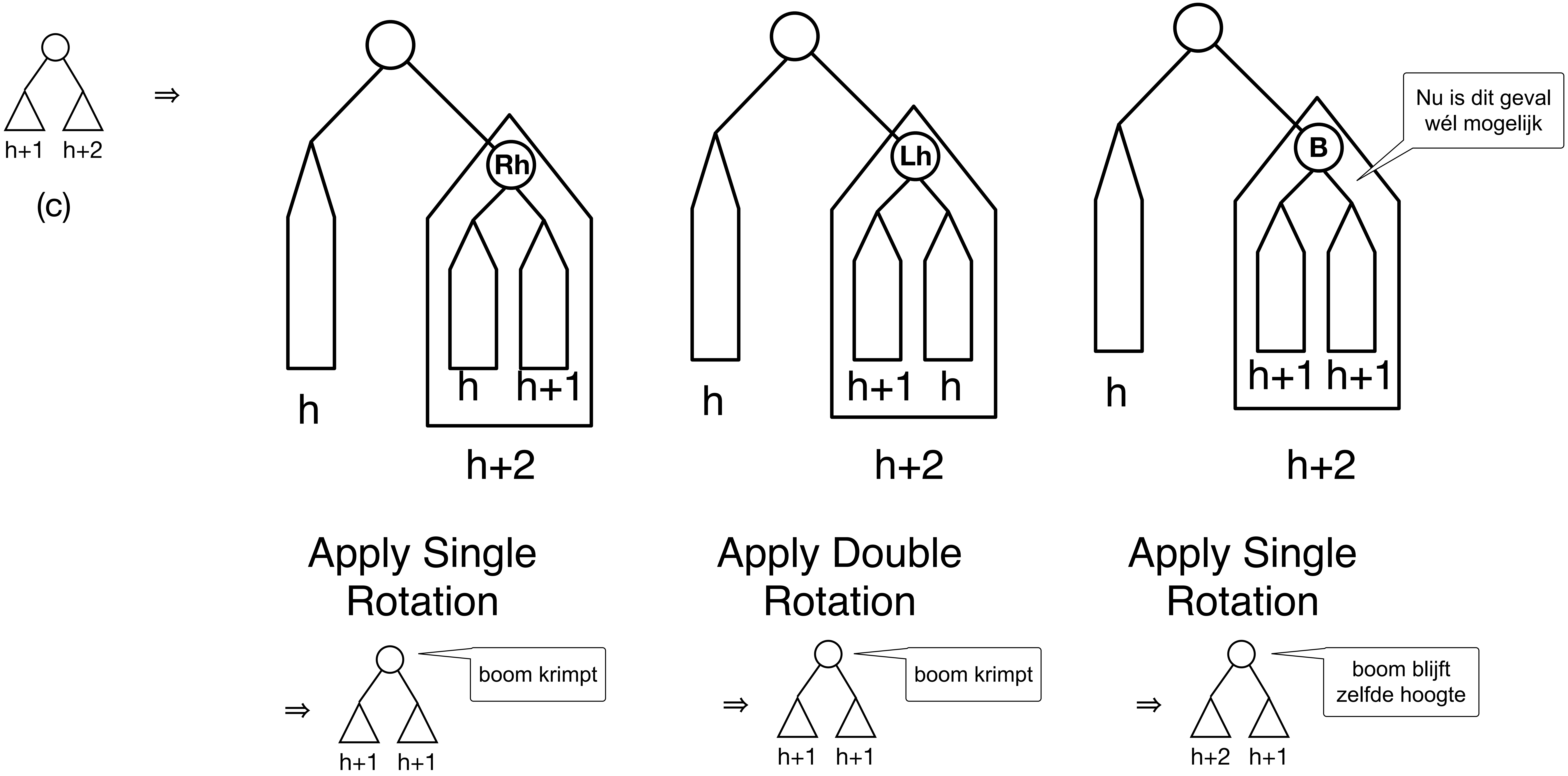
```
(define (check-after-insert-right parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Lhigh)
     (AVL-node-balance! child balanced)
     #f)
    ((eq? (AVL-node-balance child) balanced)
     (AVL-node-balance! child Rhigh)
     #t)
    (else ; child already was right-high
     (let* ((right (AVL-node-right child))
            (left (AVL-node-left right)))
       (if (eq? (AVL-node-balance right) Rhigh)
         (begin
           (child! parent (single-rotate-left! child))
           (single-rotate-left-update! child right))
         (begin
           (child! parent (double-rotate-right-then-left! child))
           (double-rotate-right-then-left-update! child right left)))
       #f))))
```

Wat kan er mislopen na delete?



We concentreren ons nu op geval (b) na (c)

Geval (b) na (c): in detail



Het delete Algoritme

```
(define (delete! bst val)
  (define <<? (lesser bst))
  (define ==? (equality bst))
  (define (find-leftmost deleted parent chil
    ...)
  (define (delete-node parent child! child)
    ...)

  (let find-node
    ((parent tree:null-tree)
     (child! (lambda (ignore child) (root! b
     (child (root bst)))
    (cond
      ((tree:null-tree? child)
       #f)
      ((==? (tree:value child) val)
       (delete-node parent child! child)
       (tree:value child))
      ((<<? (tree:value child) val)
       (find-node child tree:right! (tree:ri
      ((<<? val (tree:value child))
       (find-node child tree:left! (tree:lef
```

```
(define (delete! avl val)
  (define ==? (equality avl))
  (define <<? (lesser avl))

  (define (find-leftmost deleted parent child! child)
    ...)
  (define (delete-node parent child! child)
    ...)

  (let find-node
    ((parent null-tree)
     (child! (lambda (ignore child) (root! avl child)))
     (child (root avl)))
    (cond
      ((null-tree? child)
       #f)
      ((==? (AVL-node-value child) val)
       (delete-node parent child! child))
      ((<<? (AVL-node-value child) val)
       (if (find-node child AVL-node-right! (AVL-node-right child))
         (check-after-delete-right parent child! child)
         #f))
      ((<<? val (AVL-node-value child))
       (if (find-node child  AVL-node-left! (AVL-node-left child))
         (check-after-delete-left parent child! child)
         #f))))
  avl))
```

Het delete Algoritme (ctd.)

```
(define (find-leftmost deleted paren
  (if (tree:null-tree? (tree:left ch
    (begin
      (tree:value! deleted (tree:val
      (child! parent (tree:right chi
    (find-leftmost deleted child
                   tree:left!
                   (tree:left child)
```

```
(define (delete-node parent child! ch
   (cond
     ((tree:null-tree? (tree:left chi
      (child! parent (tree:right chil
     ((tree:null-tree? (tree:right ch
      (child! parent (tree:left chil
     (else
      (find-leftmost child
                     child
                     tree:right!
                     (tree:right child)))))
```

```
(define (delete! avl val)
  (define ==? (equality avl))
  (define <<? (lesser avl))

  (define (find-leftmost deleted parent child! child)
    (if (null-tree? (AVL-node-left child))
      (begin
        (AVL-node-value! deleted (AVL-node-value child))
        (child! parent (AVL-node-right child))
        #t)
      (if (find-leftmost deleted child AVL-node-left! (AVL-node-left child))
        (check-after-delete-left parent child! child)
        #f)))

  (define (delete-node parent child! child)
    (cond
      ((null-tree? (AVL-node-left child))
       (child! parent (AVL-node-right child))
       #t)
      ((null-tree? (AVL-node-right child))
       (child! parent (AVL-node-left child))
       #t)
      (else
       (if (find-leftmost child child AVL-node-right! (AVL-node-right child))
         (check-after-delete-right parent child! child)
         #f))))
  (let find-node
    ..))
```

Eventueel herbalanceren na links deleten

Het andere geval is symmetrisch

```
(define (check-after-delete-left parent child! child)
  (cond
    ((eq? (AVL-node-balance child) Lhigh)
     (AVL-node-balance! child balanced)
     #t)
    ((eq? (AVL-node-balance child) balanced)
     (AVL-node-balance! child Rhigh)
     #f)
    (else ; right-high
     (let* ((right (AVL-node-right child))
            (right-bal (AVL-node-balance right))
            (left (AVL-node-left right)))
       (if (or (eq? right-bal balanced)
               (eq? right-bal Rhigh))
         (begin
           (child! parent (single-rotate-left! child))
           (single-rotate-left-update! child right))
         (begin
           (child! parent (double-rotate-right-then-left! child))
           (double-rotate-right-then-left-update! child right left)))
       (not (eq? right-bal balanced))))))
```

Conclusie Dictionaries

	Gewone lijst	sorted list (vector)	sorted list (gelinkt)	BST	AVL
<u>insert!</u>					
worst	O(1)	O(n)	O(n)	O(n)	O(log(n))
average	O(1)	O(n)	O(n)	O(log(n))	O(log(n))
<u>delete!</u>					
worst	O(n)	O(n)	O(n)	O(n)	O(log(n))
average	O(n)	O(n)	O(n)	O(log(n))	O(log(n))
<u>find</u>					
worst	O(n)	O(log(n))	O(n)	O(n)	O(log(n))
average	O(n)	O(log(n))	O(n)	O(log(n))	O(log(n))

*log(n) is ondergrens **comparatief** zoeken*

Kunnen we nóg sneller?

JA!

Hoofdstuk 6

6.1 De Structuur van bomen
6.1.1 Terminologie
6.1.2 Binaire Bomen
6.1.4 Alternatieve Representaties

6.2 Het doorlopen van bomen
6.2.1 Diepte-Eerst
6.2.2 Breedte-Eerst

6.3 Binaire Zoekbomen
6.3.1 Lijstgebaseerde dictionaries
6.3.2 Binaire Zoekbomen
6.3.3 Boomgebaseerde dictionaries

6.4 AVL Bomen
6.4.1 Representatie van Nodes
6.4.2 Herbalanceren door roteren
6.4.3 Insert
6.4.4 Delete
6.4.5 Find