

# Informatica

2<sup>e</sup> semester: les 8

Binaire bomen &  
computatietheorie

Jan Lemeire

**Informatica 2<sup>e</sup> semester**  
*februari – mei 2021*



Vrije Universiteit Brussel

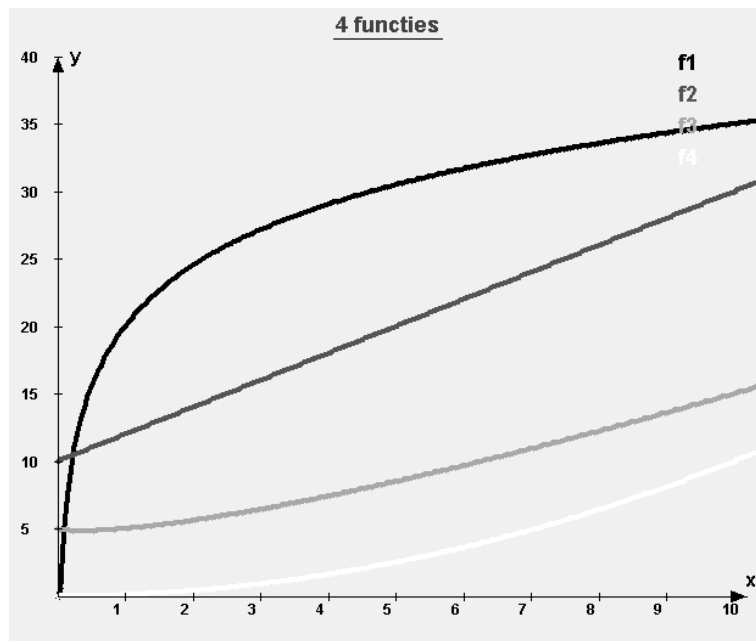
# Vandaag

- 1. Hfst 1: Grootte-orde**
- 2. Hfst 7: Binaire bomen**
- 3. Deel III – hfst 6a: computatietheorie**

# Hfst 1 Grootte-orde

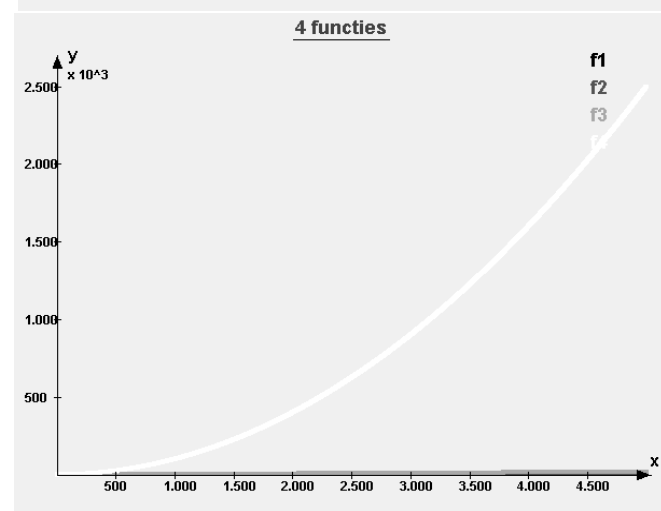
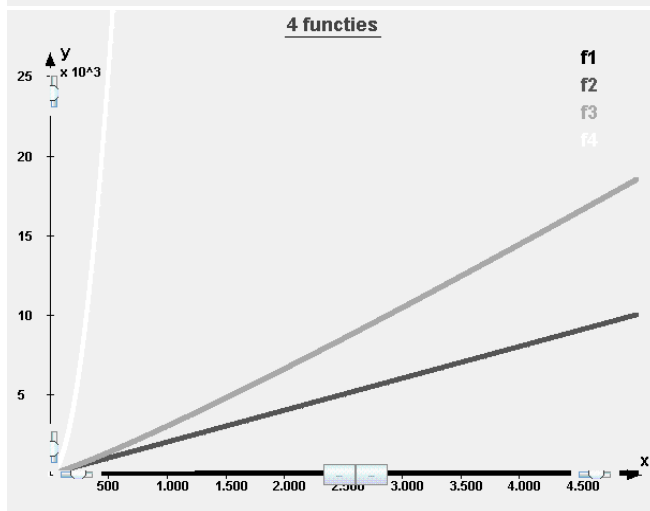
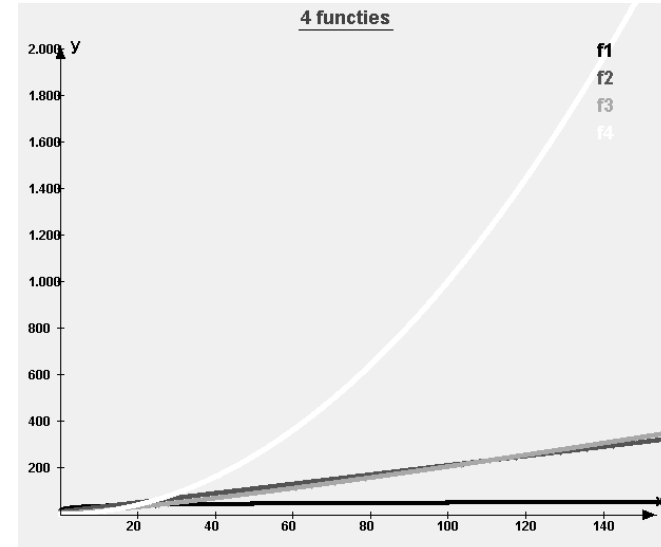
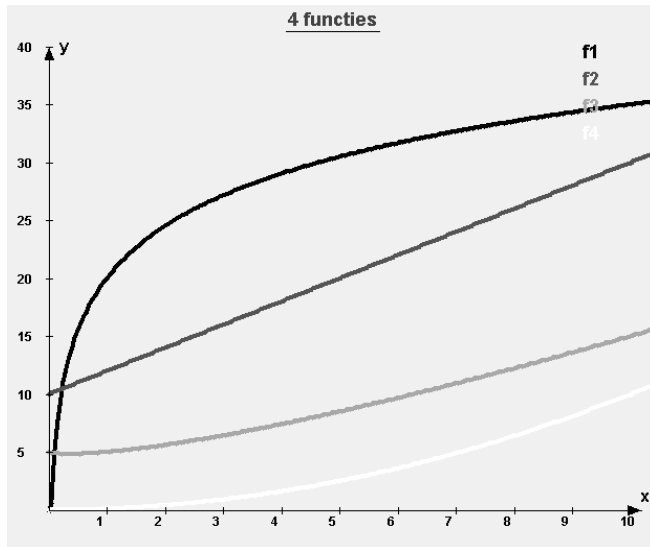
# Raadseltje

- $f1(x): y = 20 + 15 \cdot \log_{10}(x)$
- $f2(x): y = 10 + 2 \cdot x$
- $f3(x): y = 5 + x \cdot \log_{10}(x)$
- $f4(x): y = 0.1 \cdot x^2$



- Wie wint? Welke functie wordt de grootste bij grote x?
- Hoe dicht liggen de functies bij elkaar bij zeer grote x?

- $f1(x): y = 20 + 15 \cdot \log_{10}(x)$
- $f2(x): y = 10 + 2 \cdot x$
- $f3(x): y = 5 + x \cdot \log_{10}(x)$
- $f4(x): y = 0.1 \cdot x^2$



# De big-o notatie

$O(1) < O(\log(n)) \ll O(n) < O(n \cdot \log(n)) \ll O(n^2)$

*Maar ook*

$O(\text{polynomiaal}) \ll O(\text{exponentiële functie})$

Bijvoorbeeld:  $O(n^{10}) \ll O(2^n)$

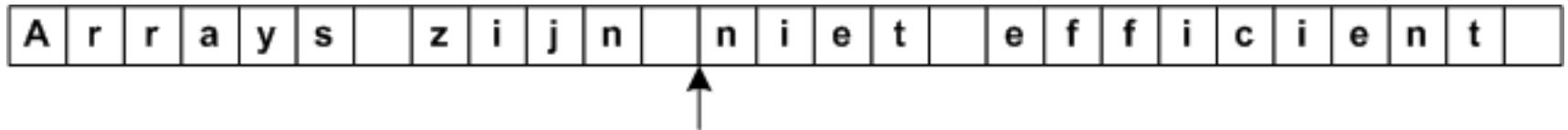
n	$n^{10}$	$2^n$
20	$20^{10} \approx 10^{13}$	$2^{20} \approx 10^6$
1000	$1000^{10} = 10^{30}$	$10^6 \cdot 10^{50} = 10^{56}$
1000000	$10^{60}$	$2^{1000000} \approx 10^6 \cdot 10^{500000} = 10^{500006}$

# Performantie datastructuren

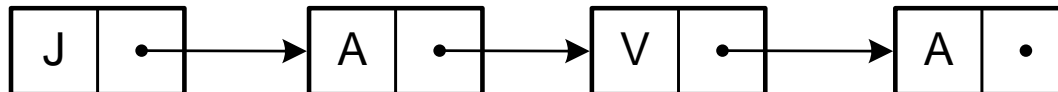
Datastructuur	Random access (opvragen i <sup>de</sup> element)	Find (via naam)	Toevoegen / verwijderen
Array	$O(0)$ ++	$O(n)$ $O(\log(n))$ als gesorteerd	$O(n)$ -
ArrayList	$O(0)$ ++	$O(n)$ $O(\log(n))$ als gesorteerd	$O(n)$ -
Linked list	$O(n)$ -	$O(n)$ --	$O(0)$ ++
Binaire boom	n.v.t.	$O(\log(n))$ +	$O(0)$ ++
Hashtabel	n.v.t.	$O(0)$ ++	$O(0)$ ++ Zolang binnen grootte

# Lineaire datastructuren

## ◆ Arrays niet flexibel

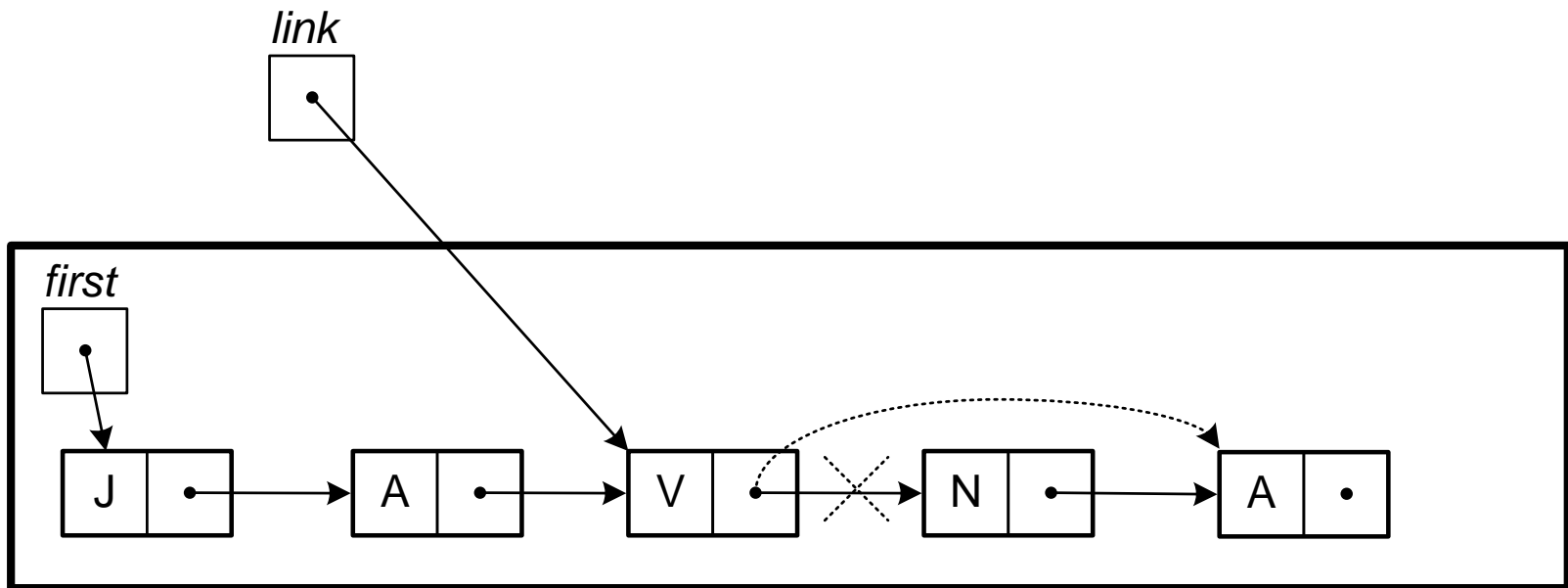


## ◆ Linked Lists: Flexibel toevoegen en verwijderen





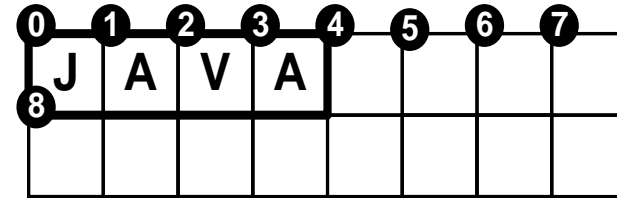
# Flexibel verwijderen



# Vinden van het $i^{\text{de}}$ element

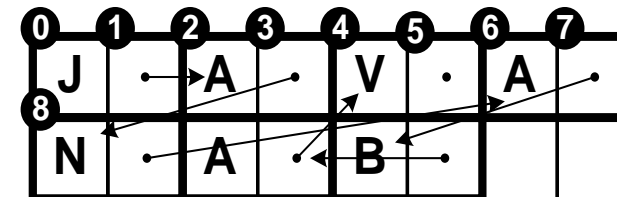
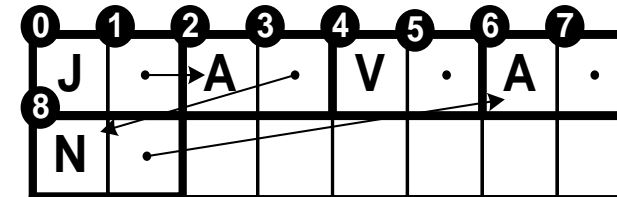
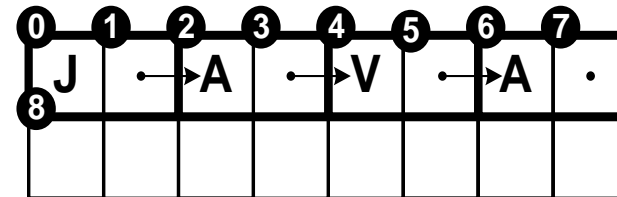
## ◆ Array

= beginvakje +  $i$



## ◆ Linked List

*Je bent van in het begin  
niet zeker waar het  
 $i$ -de element zich bevindt  
in het geheugen, zeker nadat  
elementen zijn toegevoegd  
en verwijderd*

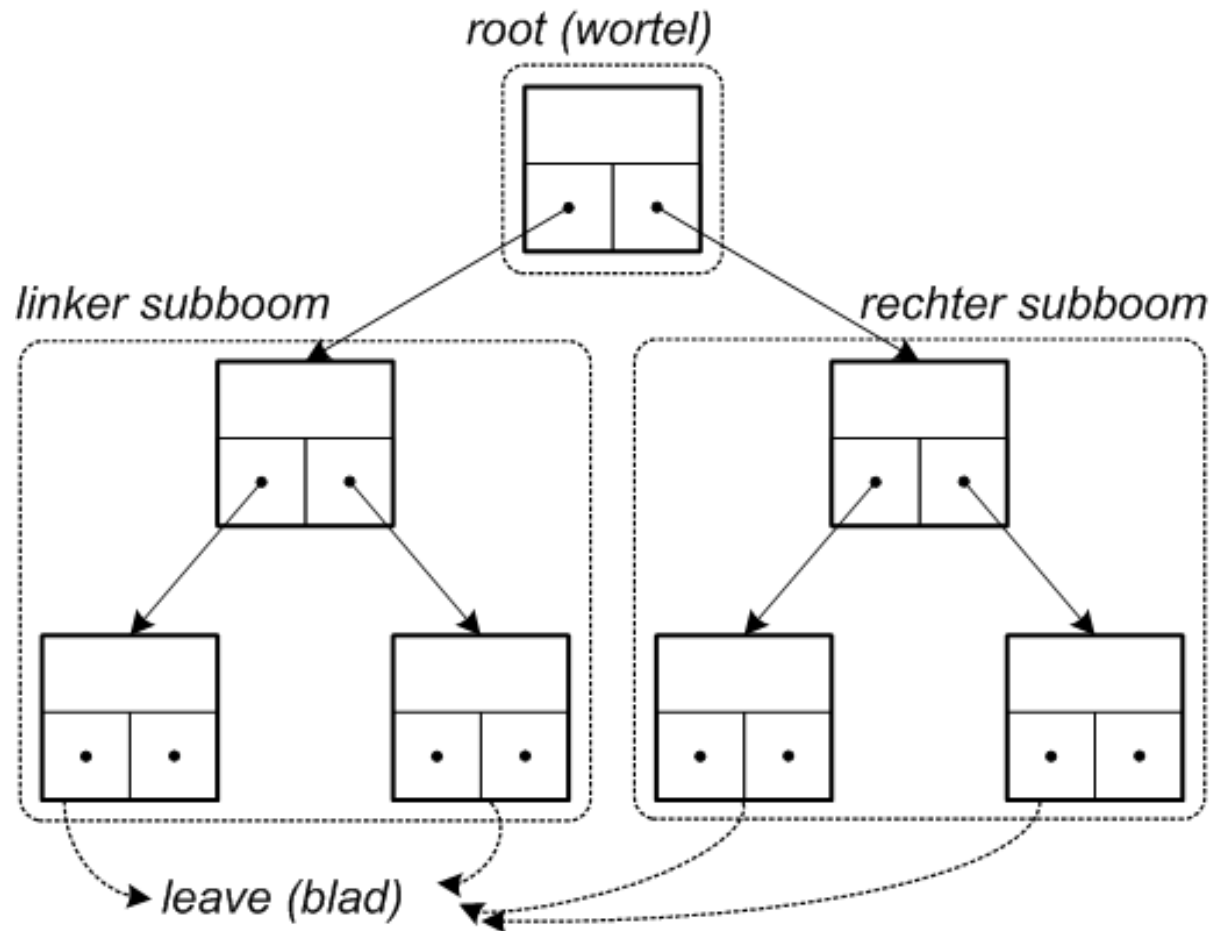


**Binaire bomen**

# Bomen (hoofdstuk 7)

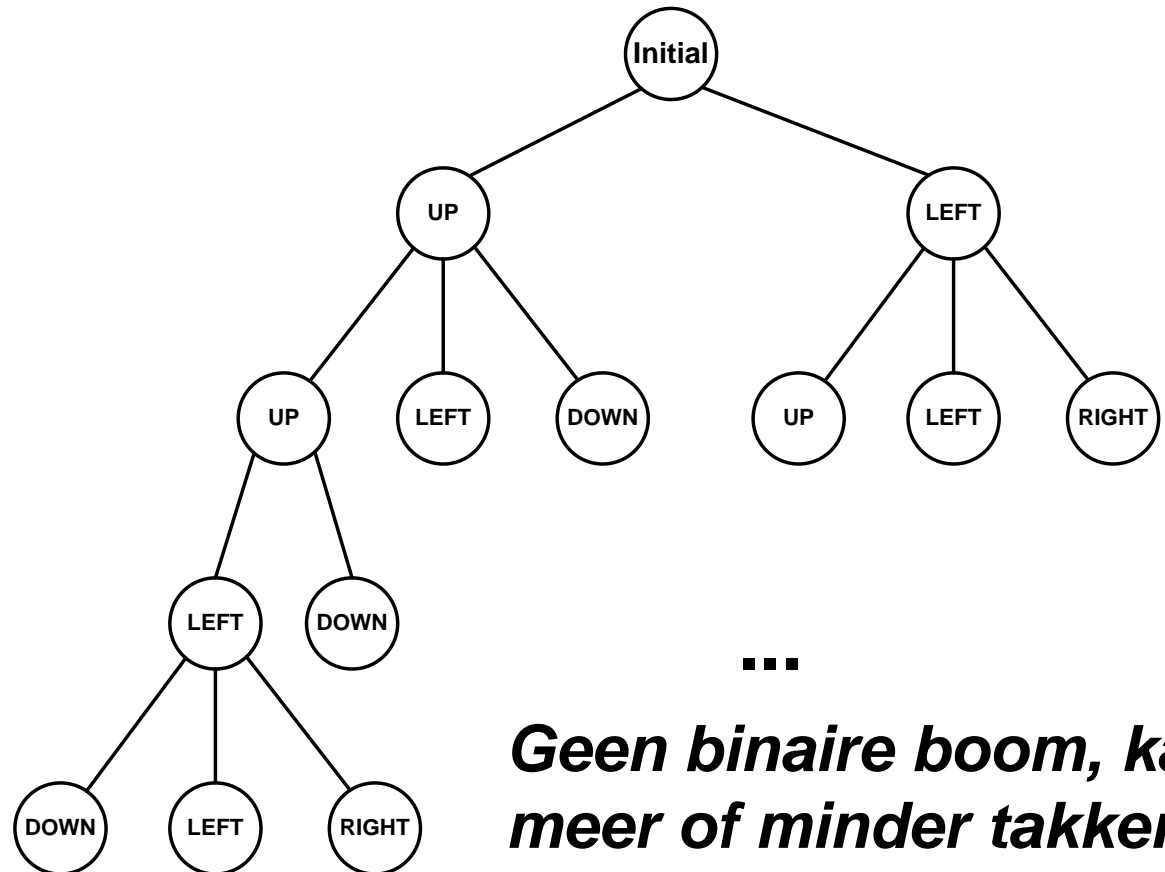
- ◆ **Arrays:** basis
- ◆ **ArrayList:** flexibele grootte van array
- ◆ **Linked Lists:**
  - ✦ flexibel toevoegen en verwijderen van elementen
  - ✦ Maar: doorlopen van lijst is traag (bvb zoeken van element)
- ➔ Nodig: flexibele structuur om elementen in op te zoeken

# Binaire boom



# Alle mogelijkheden afgaan = aanmaken boom (les 7)

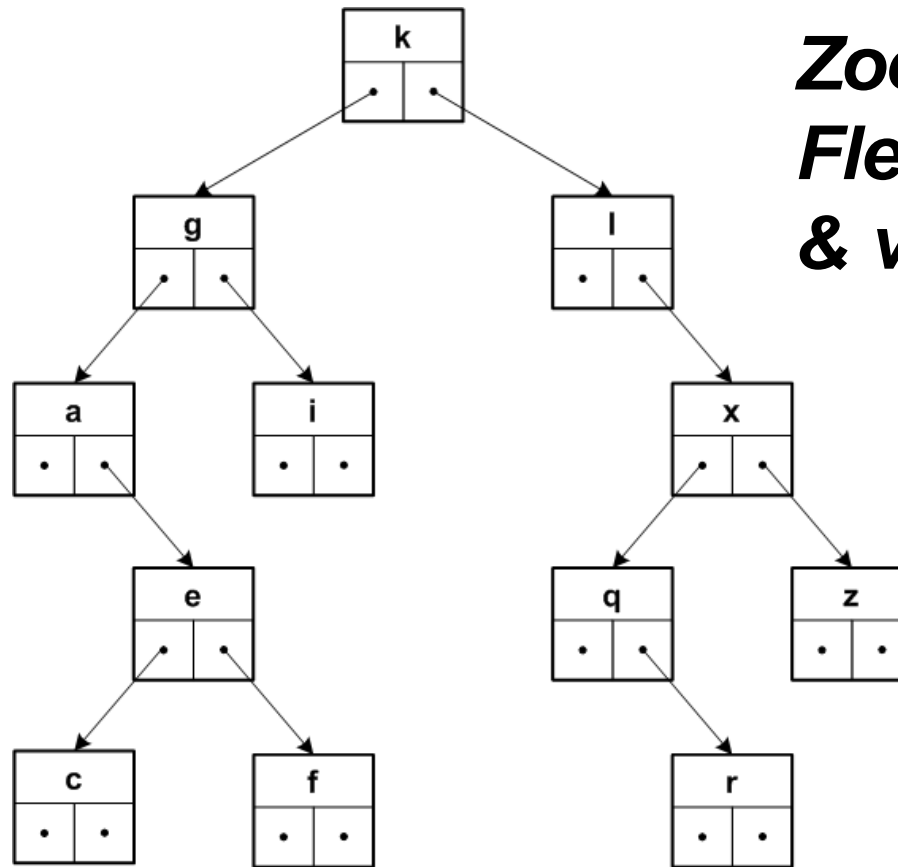
	← 5	2
↑ 1	8	3
4	7	6



***Oplossing puzzel  
zoeken***

***Geen binaire boom, kan  
meer of minder takken  
hebben***

# Geordende binaire boom



***Zoeken kan snel  
Flexibel: toevoegen  
& verwijderen***

# Elke node is een object

```
class Node<T>{  
    T data;  
    Node<T> left, right;  
  
    Node(T data) {  
        this.data = data;  
        this.left = null;  
        this.right = null;  
    }  
}
```



# Boom object

```
public class BinaryTree<T> {  
    Node<T> root;  
    Comparator<T> comparator;  
  
    public BinaryTree(Comparator<T> comparator) {  
        this.comparator = comparator;  
        root = null;  
    }  
}
```

# Definiëren van ordening

java.util

## Interface Comparator<T>

### Method Summary

int compare(T o1, T o2)

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

boolean

equals(Object obj)

Indicates whether some other object is "equal to" this comparator.

# IntegerComparator

```
class IntegerComparator implements Comparator<Integer> {  
    @Override  
    public int compare(Integer val1, Integer val2) {  
        return val1 - val2;  
    }  
}
```

# Ordenen van studenten op naam

```
class StudentComparator implements Comparator<Student>{
    @Override
    public int compare(Student student1, Student student2) {
        int ordeNaam = student1.naam.compareTo(student2.naam);
        if (ordeNaam == 0) // als beide dezelfde naam
            return student1.voornaam.compareTo(student2.voornaam);
        else
            return ordeNaam;
    }
}
```

# Aanmaken van boom

```
IntegerComparator comparator  
    = new IntegerComparator();  
  
BinaryTree<Integer> tree  
    = new BinaryTree<Integer>(comparator);
```

***korter:***

```
BinaryTree<Integer> tree  
    = new BinaryTree<Integer>(new IntegerComparator());
```

# Nog korter: via *anonymous* class

```
BinaryTree<Integer> tree =  
    new BinaryTree<Integer>(  
        new Comparator<Integer>() {  
            @Override  
            public int compare(Integer val1, Integer val2)  
            {  
                return val1 - val2;  
            }  
        }  
    );
```

# Rekursie

# Recurisie

```
private void recursion4(int t)
{
    System.out.println(t);
    t++;
    if (t < 10)
        recursion4(t);
}
```

- Bij aanroep van recursion4(6) geeft dit:

6  
7  
8  
9



# Terugkeer van de recursie

```
private int recursion5(int t){  
    if (t < 10){  
        recursion5(t+1);  
    }  
    System.out.println(t);  
}
```

- Output bij aanroep van `recursion5(6)` :

10  
9  
8  
7  
6

**Moeilijkste van  
recursie**

# Na het oproepen van een functie, ga je verder met de rest

```
void f(){  
    System.out.println("a");  
    g();  
    System.out.println("b");  
}  
void g(){  
    System.out.println("c");  
}
```

◆ Output:

a

c

b



**Na het uitvoeren van g(), wordt  
de rest van f() uitgevoerd**

# Recursie zonder recursie?

## ◆ Staartrecursie

✦ 1 recursieve oproep

Kan ook geprogrammeerd worden met een *while*

## ◆ Boomrecursie

✦ Meer dan 1 recursieve oproep (vb fibonacci)

Is minder evident om zonder recursie te programmeren,  
bij **Mergesort** zien we hoe dit kan. Aanpak zoals bij  
**breadth-first search**.

# Operaties op bomen

# Belangrijke operaties

7.3 Zoeken element

7.5 Toevoegen element (opbouwen boom)

7.6 Verwijderen element (NIET TE KENNEN)

7.7 Doorlopen van boom (bvb printen)

## 7.3. Zoeken element

```
public boolean contains(T object){
    return find(root, object);
}

private boolean find(Node<T> current, T object){
    if (current == null)
        return false;
    else if (comparator.compare(current.data, object) == 0)
        return true;
    else if (comparator.compare(object, current.data) < 0)
        return find(current.left, object);
    else
        return find(current.right, object);
}
```

◆ Tijd  $\sim$  diepte boom

◆ Idealiter:  $\log_2 n$

✦ Behalve als ongebalanceerd (zie verder)

# Staartrecursie

◆ Met while:

```
public boolean containsWithoutRecursion(T object){
    Node<T> current = root;
    while (current != null){
        if (comparator.compare(current.data, object) == 0)
            return true; // gevonden!
        else if (comparator.compare(object, current.data) < 0)
            current = current.left;
        else
            current = current.right;
    }
    return false; // niet gevonden
}
```

# 7.4 Zoektijd binaire boom

Als gebalanceerd

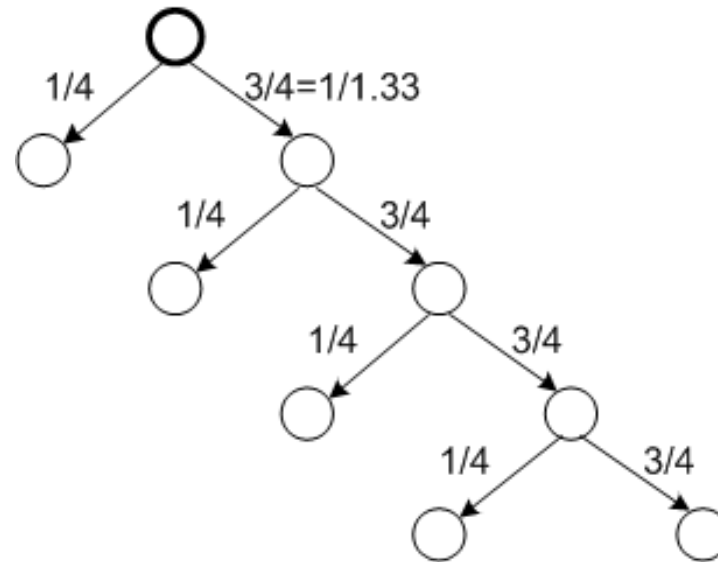
◆  $n \Rightarrow n/2 \Rightarrow n/4 \Rightarrow n/8 \Rightarrow \dots \Rightarrow 1$

◆ Of:  $1.2^x = n$

◆  $x = \text{aantalZoekstappen} = \log_2 n$



# Ongebalanceerde boom




$$\text{aantalZoekstappen} = \log_4 n$$

$$\text{aantalZoekstappen} = \log_{1.33} n$$

Gemiddelde is slechter dan  $\log_2 n$ !

# Zoektijd

	$\log_2 n$	$\log_4 n$	$\log_{1.33} n$	$\frac{n}{2k} + \log_2 k$	$n/2$
<b>10</b>	3,3	1,7	8,1	16,6	5
<b>100</b>	6,6	3,3	16,1	16,6	50
<b>1000</b>	10,0	5,0	24,2	16,6	500
<b>10000</b>	13,3	6,6	32,3	16,7	5000
<b>100000</b>	16,6	8,3	40,0	17,1	50000
<b>1000000</b>	19,9	10,0	48,4	21,6	500000
<b>10000000</b>	23,3	11,6	56,5	66,6	5000000
<b>100000000</b>	26,6	13,3	64,6	516,6	50000000
<b>1000000000</b>	29,9	14,9	72,7	5016,6	500000000

  
/2

 \*2,433

# Verskil in zoektijd

◆ Eigenschap logaritmes :  $\log_d n = \frac{\log_2 n}{\log_2 d}$  Verandering van basis

◆ Verschil in aantal stappen:

$$stappenRatio = \frac{\log_d n}{\log_2 n} = \frac{1}{\log_2 d}$$

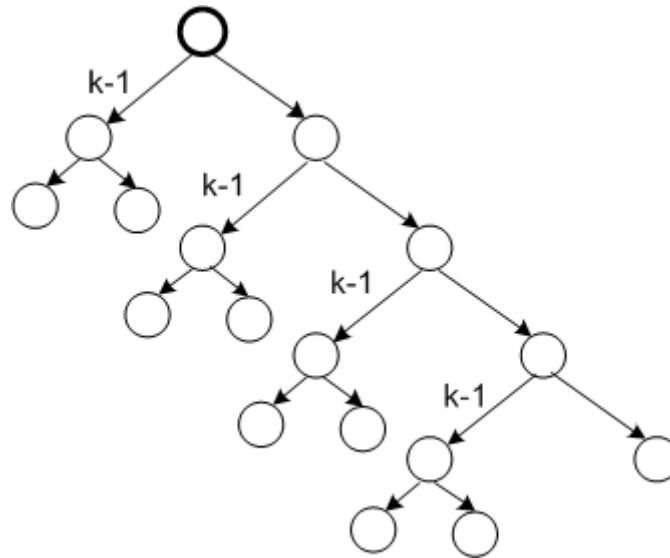
$$stappenRatio = \frac{1}{\log_2 1.33} = \frac{1}{0.411} = 2.43 \quad 2 \text{ t.o.v. } 1.333$$

$$stappenRatio = \frac{1}{\log_2 4} = 1/2 \quad 2 \text{ t.o.v. } 4$$

◆ Besluit: logaritmisch  $\ll$  lineair

# Ongebalanceerde boom 2

- ◆ Vast aantal  $(k-1)$  nodes links, de rest rechts



$$k \ll n$$

- ◆ Rechterboom heeft  $n/k \Rightarrow n/2k$  stappen
- ◆ Totaal  $n/2k + \log_2(k-1) \approx n/2k$  **lineair**

## 7.5. Toevoegen element

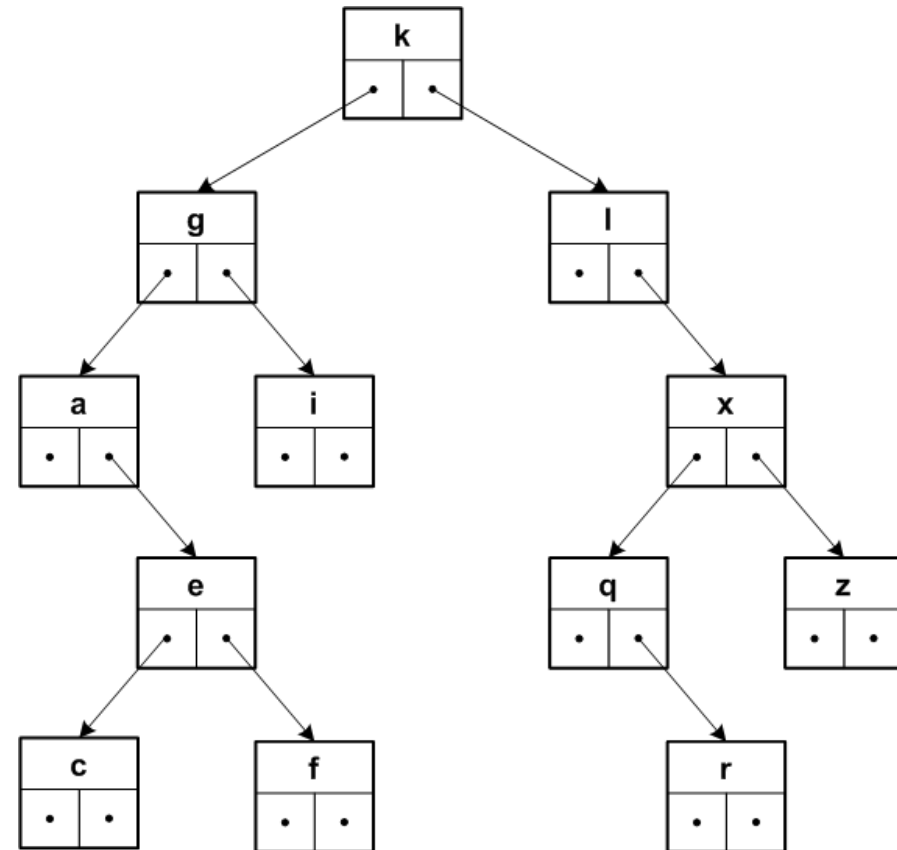
```
public void add(T object) {
    Node<T> newNode = new Node<T>(object);
    if (root == null)
        root = newNode;
    else
        add(root, newNode);
}

private void add(Node<T> current, Node<T> newNode) {
    if (comparator.compare(newNode.data, current.data) < 0) {
        // moet links komen
        if (current.left == null)
            current.left = newNode;
        else
            add(current.left, newNode);
    } else {
        // rechts
        if (current.right == null)
            current.right = newNode;
        else
            add(current.right, newNode);
    }
}
```

# Volgorde van toevoegen bepaalt boom!

*Wat als ik toevoeg in die volgordes?...*

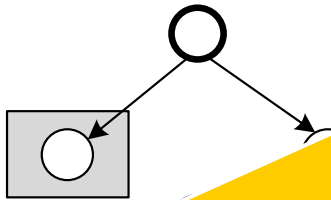
- ◆ {k, l, g, i, x, q, r, a, e, f, c, z}
- ◆ {a, c, e, f, g, i, k, l, q, r, x, z}
- ◆ {i, z, q, l, k, f, e, g, c, x, a, r}



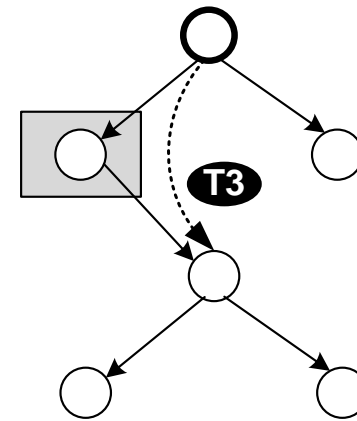
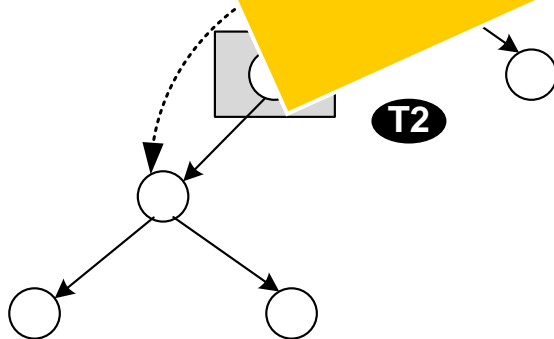
# 7.6. Verwijderen van element

Node die we verwijderen: 

- ◆ Eerste mogelijkheid: node heeft geen kinderen



- ◆ Tweede mogelijkheid: node heeft 1 subboom



**7.6 niet te kennen**

# 3<sup>e</sup> geval: node heeft 2 subbomen

Verwijderen node 3

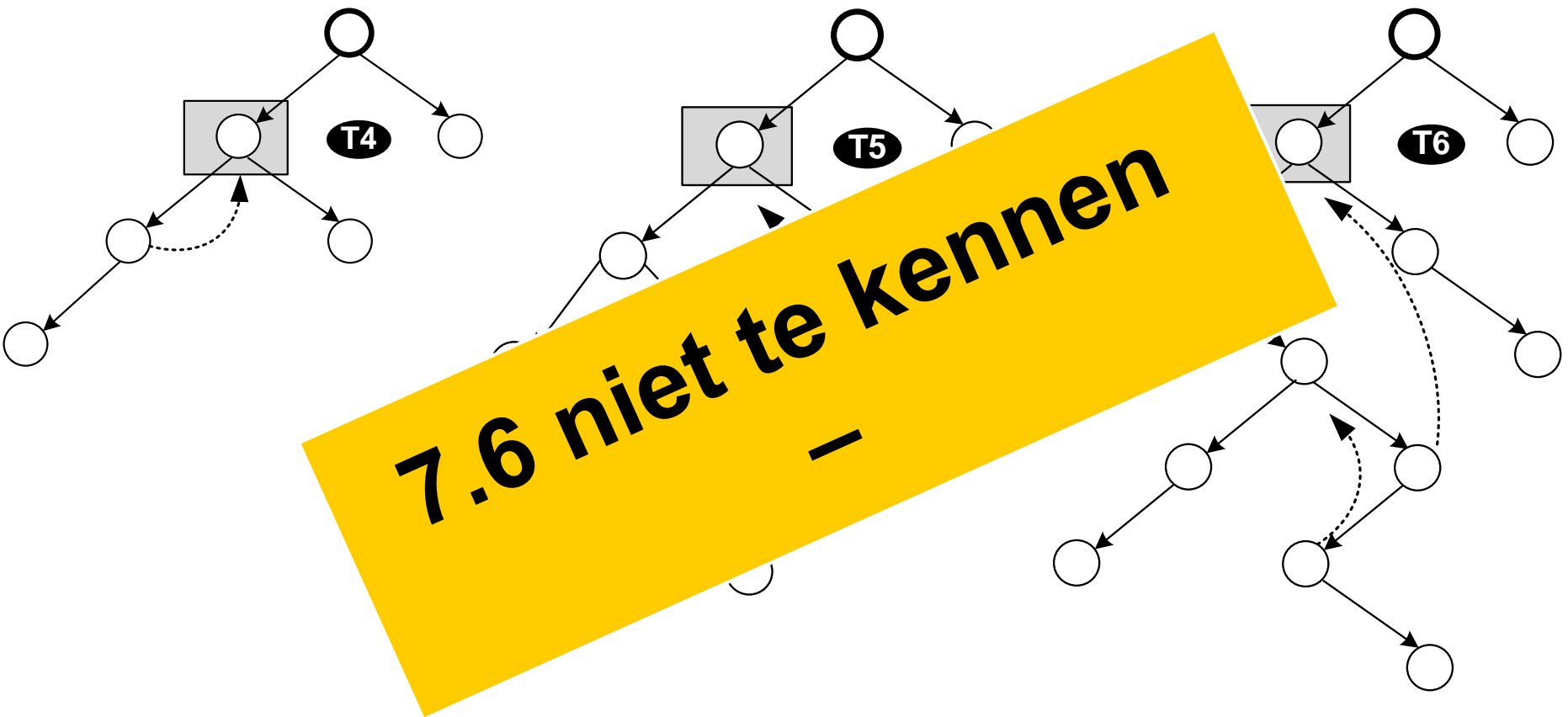


Onze keuze



# Promoten meest rechtse van linkersubboom

Niet in cursus



```

public boolean remove(T object){
    if (comparator.compare(root.data, object)== 0){
        root = createSubtree(root);
        return true;
    } else
        return findNodeAndRemove(root, object);
}

```

```

private boolean findNodeAndRemove(Node<T> current, T object){
    if (current == null)
        return false;

    if (current.left != null & comparator.compare(current.left.data,
object)== 0){
        current.left = createSubtree(current.left);
        return true;
    } else
        if (comparator.compare(current.right.data, object)== 0){
            current.right = createSubtree(current.right);
            return true;
        } else if (comparator.compare(current.data, object) < 0)
            return findNodeAndRemove(current.left, object);
        else
            return findNodeAndRemove(current.right, object);
}

```

**7.6 niet te kennen**

no longer stoppen!



```

private Node<T> createSubtree(Node<T> nodeToBeRemoved) {
    if (nodeToBeRemoved.left == null) {
        // right is the only subtree that we should consider
        return nodeToBeRemoved.right;    T1    T3
    } else if (nodeToBeRemoved.right == null) {
        // left is the only subtree that we should consider
        return nodeToBeRemoved.left;
    } else {
        // promote rightmost node
        if (nodeToBeRemoved.left != null) {
            nodeToBeRemoved.left = createSubtree(nodeToBeRemoved.left.right);    T4
            return nodeToBeRemoved.left;
        } else {
            nodeToBeRemoved.left = nodeToBeRemoved;
            return nodeToBeRemoved.left;    T5    T6
        }
    }
}

```

**7.6 niet te kennen**

```

private Node<T> pickRightMostNode(Node<T> current) {
    // we expect current.right not to be null
    if (current.right.right != null) {
        T5      return pickRightMostNode(current.right.right);
    } else {
        T6      Node<T> rightMostNode = current.right;
                current.right = rightMostNode.right;
                rightMostNode.right = null;
    }
}

```

**7.6 niet te kennen**

—  
 links van **rightMostNode**  
 gaat de plaats innemen  
 van **rightMostNode** in de boom

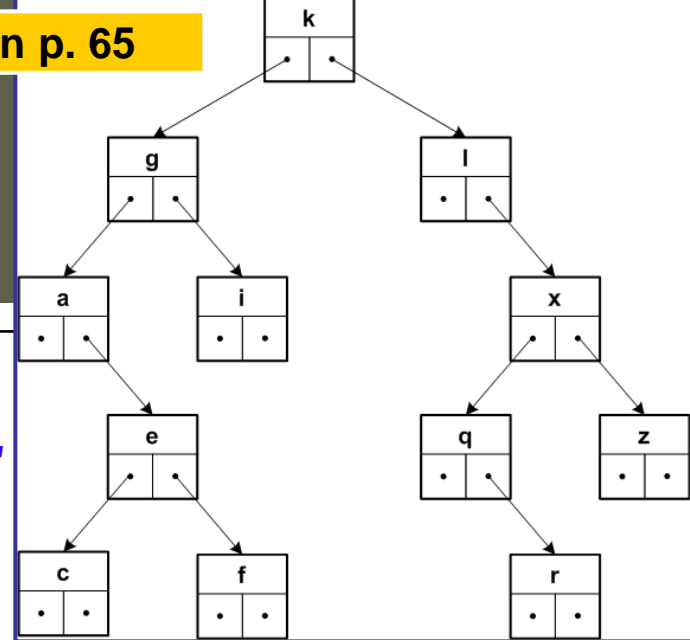
# Oefening in boek (p. 75)

We verwijderen eerst element  
'l', 'q', 'g', vervolgens 'x'  
en tot slot verwijderen we 'k'  
Teken telkens de nieuwe  
boom



# 7.7. Doorlopen van boom

```
public void preOrder(Node<T> node) {
    if (node != null) {
        System.out.print(node.data+", ");
        preOrder(node.left);
        preOrder(node.right);
    }
}
```



```
public void inOrder(Node<T> node) {
    if (node != null) {
        inOrder(node.left);
        System.out.print(node.data+", ");
        inOrder(node.right);
    }
}
```

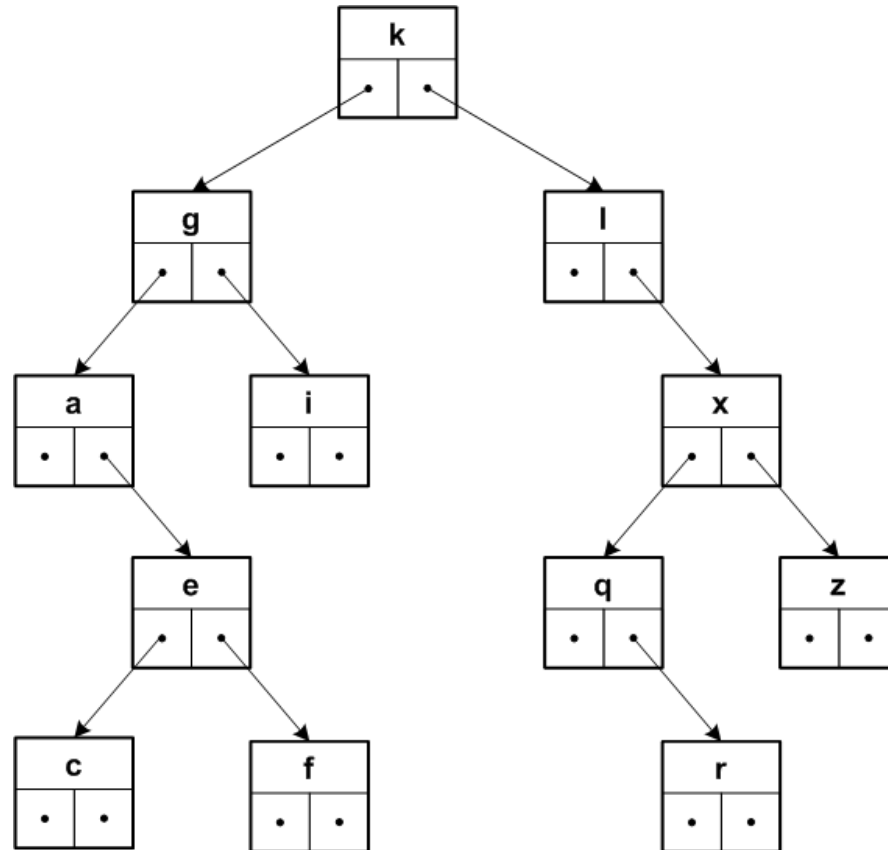
➡ {k, g, a, e, c, f, i, l, x, q, r, z}

➡ {a, c, e, f, g, i, k, l, q, r, x, z}

```
public void postOrder(Node<T> node) {
    if (node != null) {
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.data+", ");
    }
}
```

➡ {c, f, e, a, i, g, r, q, z, x, l, k}

# Boom van p.65

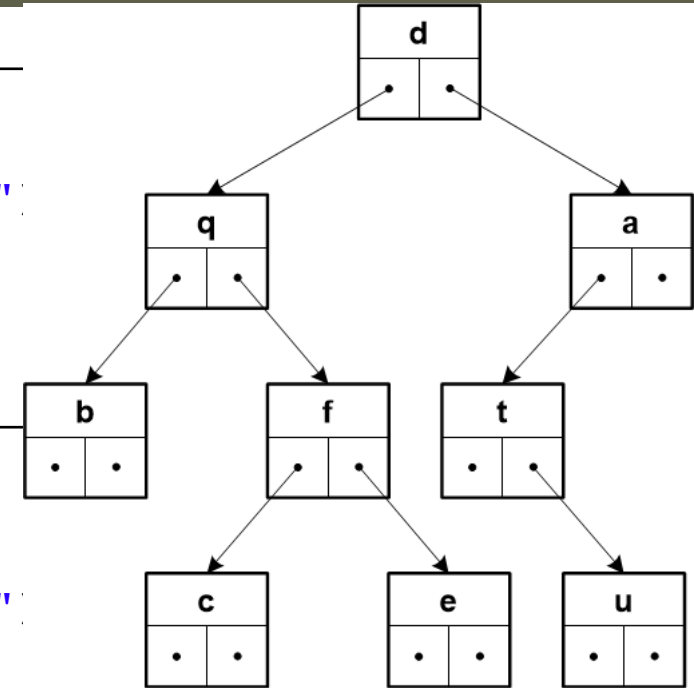


# Oefening: print de boom in de 3 volgordes

```
public void preOrder(Node<T> node) {
    if (node != null) {
        System.out.print(node.data+", ");
        preOrder(node.left);
        preOrder(node.right);
    }
}
```

```
public void inOrder(Node<T> node) {
    if (node != null) {
        inOrder(node.left);
        System.out.print(node.data+", ");
        inOrder(node.right);
    }
}
```

```
public void postOrder(Node<T> node) {
    if (node != null) {
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.data+", ");
    }
}
```



Opgelet: deze boom is niet geordend,  
inOrder() zal dus geen alfabetische lijst geven



# Toepassingen

## ◆ **preOrder:** backtracking

- ✦ Is de node de oplossing?
- ✦ Indien niet, zoek eerst links en daarna rechts

## ◆ **inOrder:** in volgorde afgaan van alle elementen

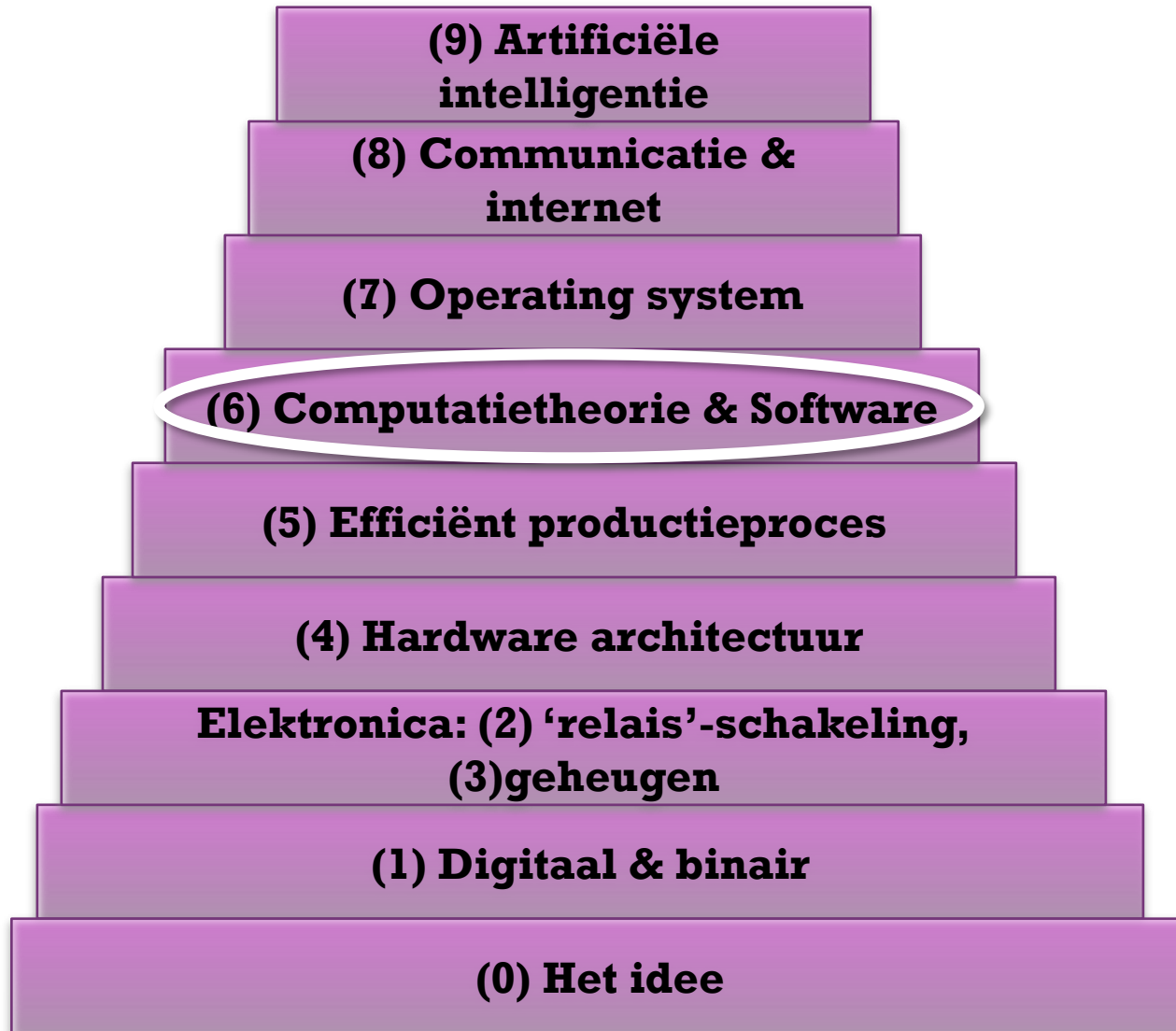
# Toepassing *postorder*

```
public int aantalKinderen(Node<T> node){  
    if (node == null)  
        return 0;  
    int n = aantalKinderen(node.left);  
    n += aantalKinderen(node.right);  
    System.out.println("Node "+node.data+" heeft "+n+"  
kinderen");  
    return n + 1;  
}
```

***Voorbeeld van boomrecursie***

***waarbij resultaat gegenereerd wordt bij het terugkeren uit de recursie***

# Waarmaken van Leibniz's droom



- 1. Kan je hetzelfde met je smartphone als met je PC/laptop? Zijn de computers evenwaardig?**
- 2. Kan je alle mogelijke programma's runnen op je smartphone/PC/laptop?**
  - Theoretisch**
  - Praktisch (compatibiliteit)**



# Hoofdstuk 6a: Computatietheorie



# Leibniz' droom



1646 – 1716

## ◆ De “**Calculus ratiocinator**”

- ◆ Een logisch denkend apparaat
- ◆ Om met de regels van de logica ondubbelzinnig te kunnen vaststellen of een statement waar of vals is
- ◆ Deductief systeem waarin alle regels afgeleid zijn van een kleine verzameling axiomas

Leibniz was één van de eerste die begreep dat als je filosoferen, denken, redeneren, ... in formele regels kunt gieten, je dit ook door een apparaat kunt laten doen.



# Gottlob Frege

1848 – 1925



- ◆ Beschrijft hoe wiskundigen en logici denken
- ◆ 1879: *de universele taal van de logica*
- ◆ 1903: brief van Bertrand Russel die hem op een onvolledigheid wijst
  - ✦ Extra-ordinaire verzameling = verzameling dat een element is van zijn eigen
    - Vb: “verzameling van alle dingen die geen spreekw zijn”
  - ✦ *Maar*: de verzameling **E** van alle ordinaire verzamelingen
    - En wat is **E**? Ordinair of extra-ordinair

Eind 19e eeuw dacht men dat alle natuurwetten binnen handbereik waren (**positivisme**). Zo begreep men met de Maxwell-vergelijkingen electriciteit en magnetisme. De relativiteitstheorie van Einstein en de kwantummechanica maakte een einde aan dit optimisme in de fysica. Maar ook in de logica bleek niet zo iets te bestaan als een mooie verzameling regels die niet tot contradicties leidt.





# David Hilbert



1862 – 1943

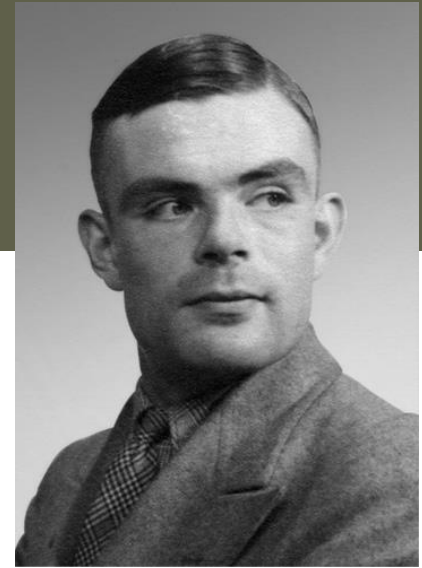
- ◆ “Wir müssen wissen; wir werden wissen”
  - ✦ We moeten weten; we zullen weten
- ◆ Elke logische of mathematische vraag is oplosbaar (en zal opgelost worden)
- ◆ *Nodig*: een expliciete procedure om vanuit premises na te gaan of een conclusie volgt of niet (Hilbert’s **beslissingsprobleem**)

Hilbert’s beslissingsprobleem zou leiden tot een doorbraak in de theoretische informatica.





# Alan Turing

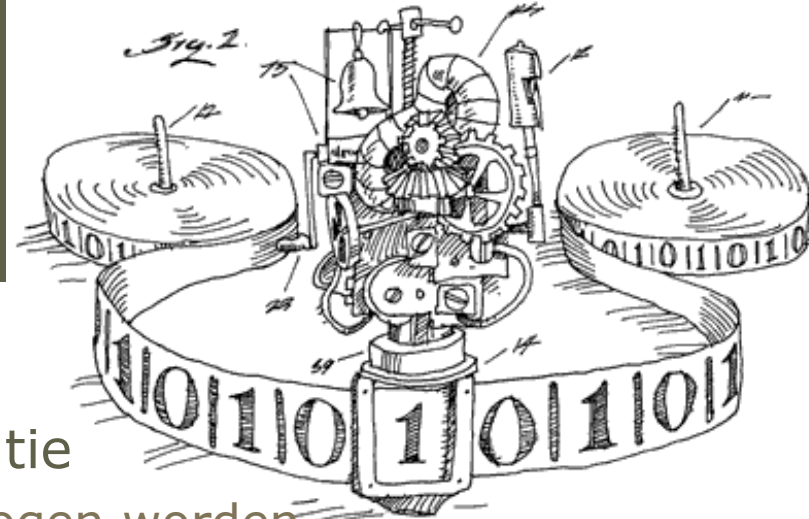


- ◆ Hilbert's procedure: **algoritme!**
- ◆ Als we een 'mechanische' lijst van regels hebben om de oplossing van een mathematisch probleem op te lossen, zouden wiskundigen geen werk meer hebben...
- ◆ Bestaat er zo'n algoritme?
- ◆ Om tegendeel te bewijzen moest Turing een machine maken dat een algoritme kan uitvoeren

Alan Turing wierp zich op Hilbert's probleem. Hij bedacht dat er hiervoor een algoritme gemaakt moest worden (die nagaat of een statement klopt of niet), en vervolgens een machine die het algoritme kan uitvoeren. De 'Turing machine' werd geboren, een theoretisch model van een computer.



# De Turing machine



- ◆ **Data:** oneindige lange tape voor het lezen of schrijven van 0/1/spatie
  - ✦ Kan 1 vakje naar links of rechts bewogen worden
- ◆ **Staat (toestand):** een aantal binaire variabelen
- ◆ **Gedragsregels:**

Staat	Gelezen bit	Nieuwe staat	Te schrijven bit	Bewegen
0	0	0	0	>
0	1	1	1	<
1	0	1	—	>
1	1	HALT		

Deze machine is alleen maar van theoretisch nut. Ze is bijvoorbeeld niet gemakkelijk programmeerbaar.



# Oplossing van Hilbert's beslissingsprobleem?

- ◆ Met de machine nagaan of een logisch statement waar of vals is:
    - ◆ Programma runnen
    - ◆ Als de machine stopt is statement bewezen (waar)
    - ◆ Je moet dus te weten komen of het programma ooit zal stoppen
  - ◆ '**Halting problem**' (tegelijkertijd ontdekt door Alonzo Church en Alan Turing)
    - ◆ Gegeven een Turing machine en programma, ga na of dit programma zal stoppen.
    - ◆ Hier bestaat **geen** algoritme voor die dit kan nagaan
- ➔ *Hilbert's beslissingsprobleem is onoplosbaar*

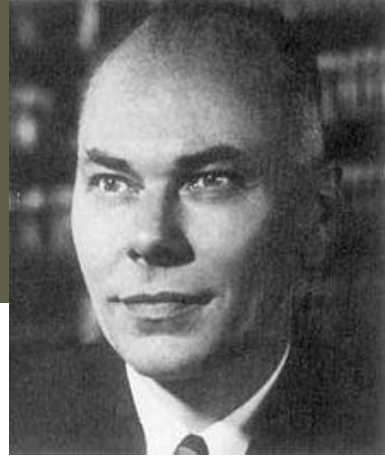
Het resultaat van Turing was weer een knak voor het optimisme van de wetenschap. Je kan niet van alles bewijzen of het waar of niet waar is. Hieruit onstond mede de moderne relativistische kijk op de wereld. De wetenschap kan niet alles oplossen/verklaren... De VUB blijft echter geloven dat wetenschap veel kan oplossen.  
**Scienta Vincere Tenebras!**

# De universele computer

- ◆ Elk algoritme kan er op uitgevoerd worden = Universele Turing Machine
- ◆ Kan alles berekenen wat berekenbaar is
  - ◆ Rekenen, maar ook om logisch te denken...
    - ➔ Universeel
- ◆ Als je met een andere computer deze computer kan simuleren, is hij ook universeel
  - ➔ Turing bewees de gelijkwaardigheid van computers

Turing's resultaten blijken wel belangrijk voor de informatica. Misschien wel het belangrijkste resultaat van Turing is het idee van een universele computer. Eén computer (hardware) die alles kan berekenen wat maar te berekenen valt. Dit was revolutionair, zeker omdat velen in die tijd niet konden geloven dat je met 1 machine totaal verschillende berekeningen kan doen. Alsof je met een wasmachine ook kan autorijden.

# Misvattingen omtrent computers



- ◆ Howard Aiken, bouwde de eerste computer voor IBM in 1944 en zei in 1956:

*"If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence I have ever encountered"*

- ◆ Studie van 1947:

*"slechts zes elektronische digitale computers zullen voldoende zijn om al het rekenwerk van de gehele Verenigde Staten te kunnen doen."*

# Computatietheorie: bewijzen dat een programma correct is...

- ◆ Heel omslachtig!
- ◆ Enkel mogelijk voor eenvoudige algoritmes.
- ◆ *Dus*: ad-hoc testen noodzakelijk, zorgvuldig programmeren, goed opdelen, ...
- ◆ Maar: belangrijk! Denk maar aan zelfrijdende autos kerncentrales of satelieten: weten dat de software correct werkt en nooit crasht.

Anderzijds zijn er nog vele gaten in de theoretische informatica. Zo zou het toch handig zijn om te bewijzen dat je programma correct is. Dit blijkt heel omslachtig te zijn, zelfs voor simple programma's. Informatica blijft dan ook gedeeltelijk een 'praktische' wetenschap. Maar wij ingenieurs malen daar toch niet om, he? Als het maar nuttig is, die computer.