

Twaalfde college algoritmiek

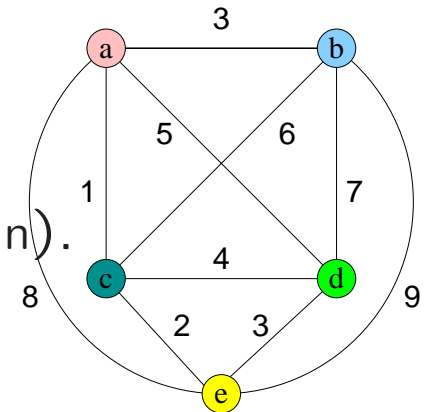
13 mei 2016

Branch & Bound

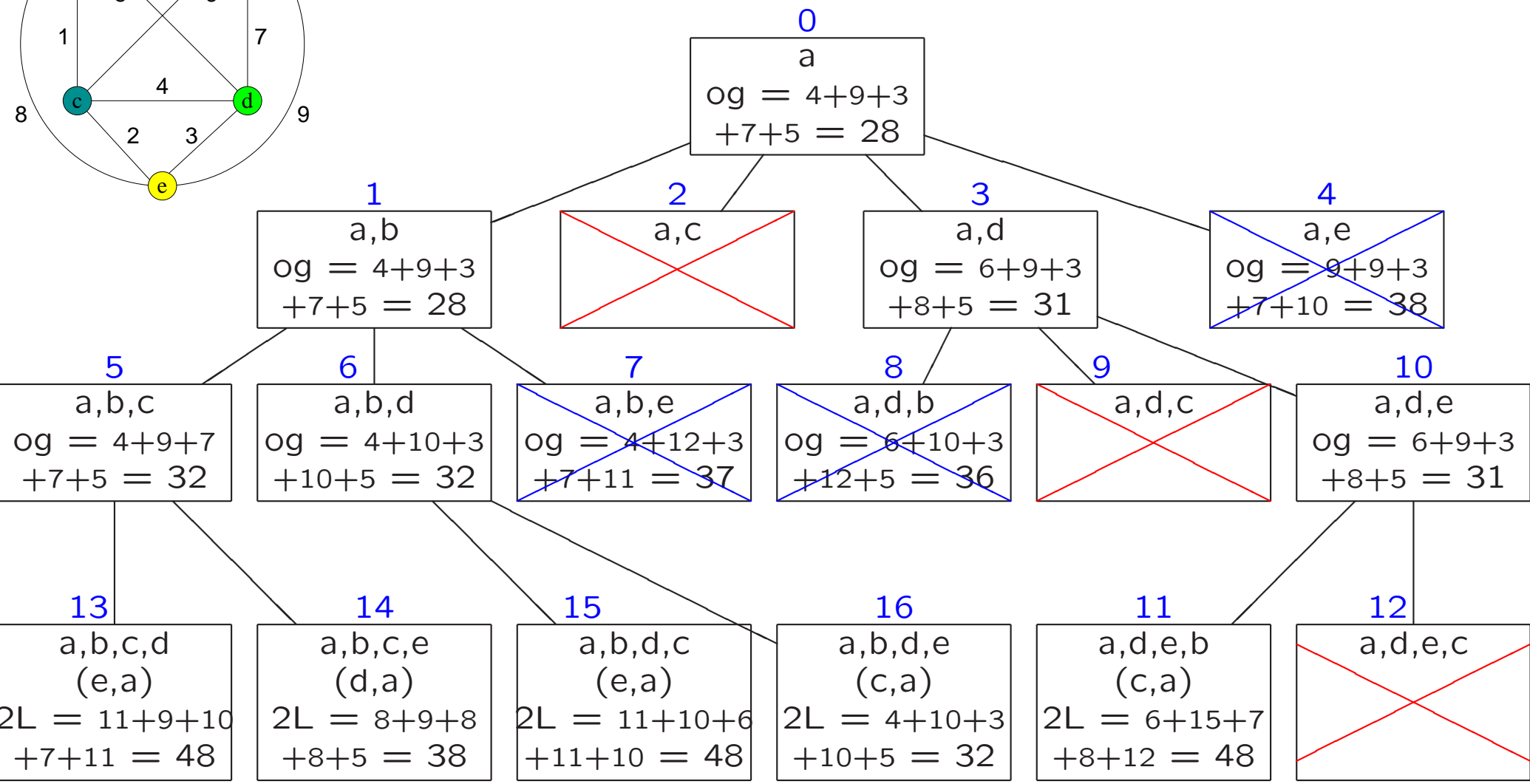
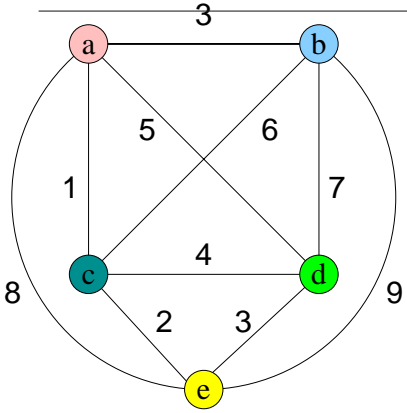
Heap, Heapsort & Heapify

Mogelijke ondergrenzen voor de kosten van een optimale oplossing:

- eenvoudig: $n \times$ kortste afstand tussen twee knopen;
 $5 \times 1 = 5$ bij aanvang (analoog als reeds takken gekozen zijn).
- iets beter: de lengtes van de n kortste takken gesommeerd;
 $1 + 2 + 3 + 3 + 4 = 13$ bij aanvang (analoog als reeds takken gekozen zijn).
- nog beter: som over alle knopen i van de afstanden van knoop i tot de twee dichtstbijzijnde knopen (inclusief al gekozen takken), en dat gedeeld door 2(*);
 $((1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3))/2 = 14$ bij aanvang.
 Opmerking: delen door 2 is niet nodig; je kunt ook $2 \times$ lengte minimaliseren (*).



tree



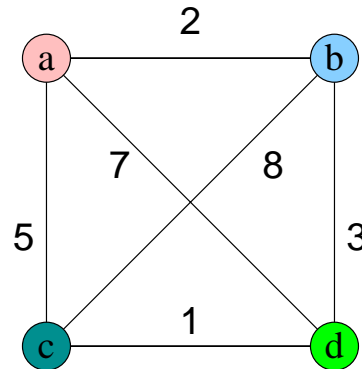
We zeiden wel:

delen door 2 is niet nodig; je kunt ook $2 \times$ lengte minimaliseren (*)

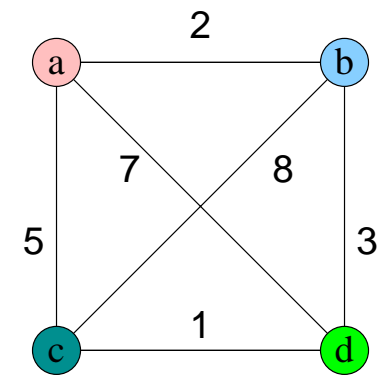
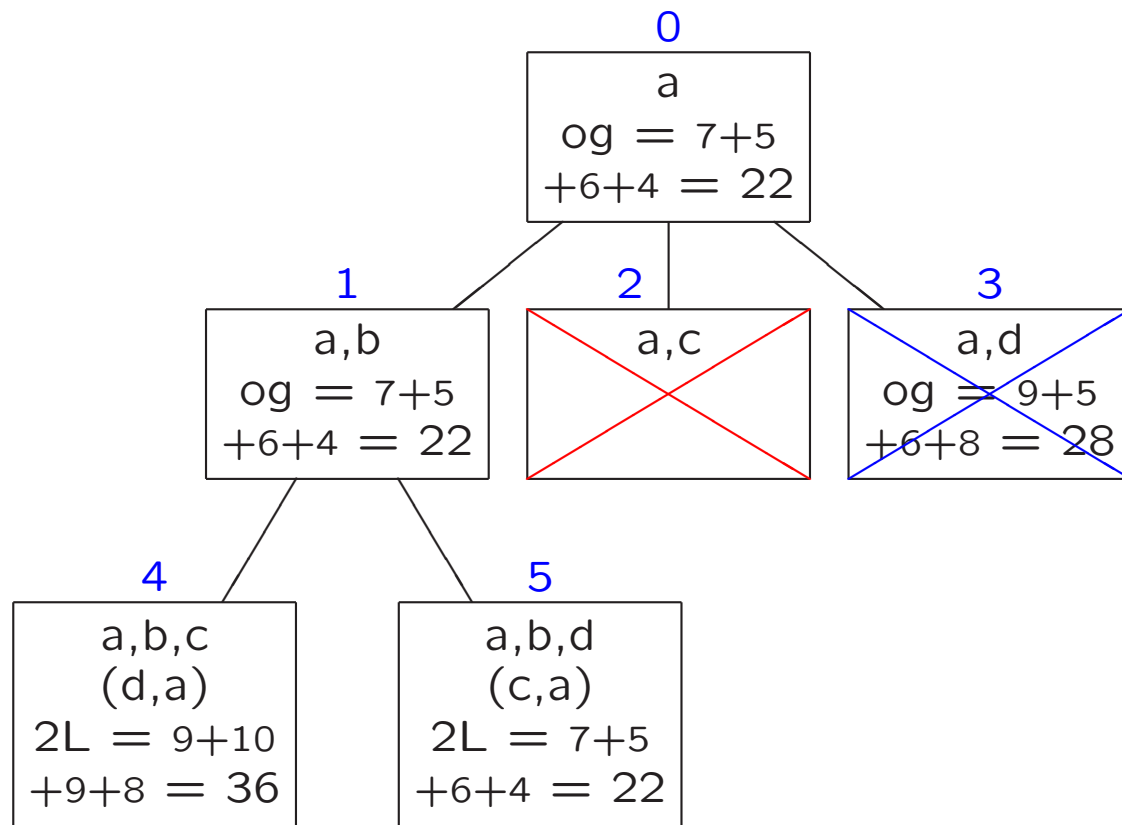
Delen door 2 geeft echter wel scherpere ondergrens, als je resultaat afrondt naar boven.

In vorige voorbeeld wordt ondergrens 31 van knoop a,d hiermee 16. Deze knoop hoef je nu niet uit te werken (zie boom in boek).

Nog een voorbeeld, met als ondergrens wederom de som van de twee kortste takken per knoop (eventueel gedeeld door 2, zoals in het boek).



De optimale oplossing heeft lengte 11: a, b, d, c, a.



Overeenkomsten en verschillen tussen:

- Exhaustive search (ES)
- Backtracking (Bt)
- Best fit first branch and bound (B&B)

N.B.: Verschillende punten hieronder hangen soms met elkaar samen.

- **ES** geschikt voor optimalisatieproblemen en andere problemen
- **Bt** geschikt voor optimalisatieproblemen en andere problemen
- **B&B** alleen geschikt voor optimalisatieproblemen

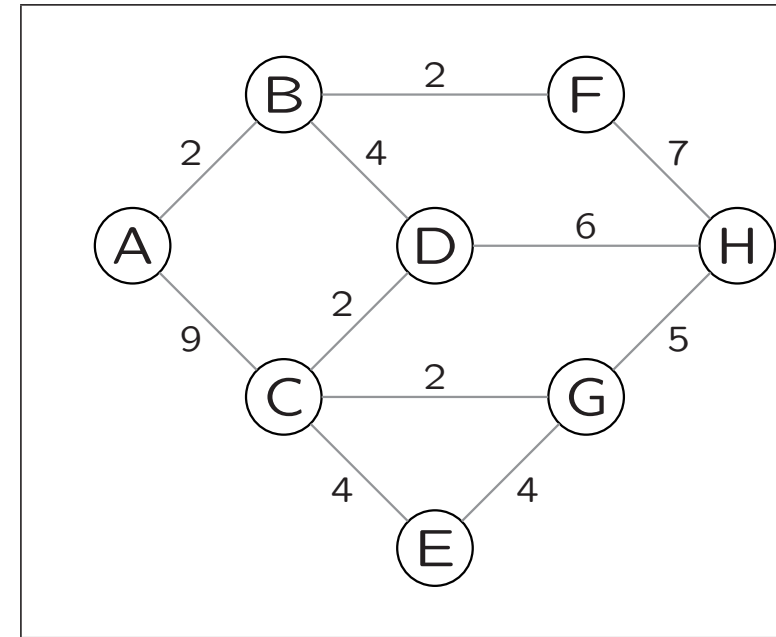
- **ES** kán oplossing component voor component opbouwen, maar is ook geschikt voor problemen waar geen sprake is van componenten en deeloplossingen
Bt bouwt oplossing component voor component op
B&B bouwt oplossing component voor component op
- **ES** controleert pas op geldigheid als het complete oplossing heeft opgebouwd (in geval dat ES oplossing component voor component opbouwt, ook verderop)
Bt breidt deeloplossing niet verder uit als het geen geldige complete oplossing meer kan worden
B&B breidt deeloplossing niet verder uit als het geen geldige complete oplossing meer kan worden

- **ES** let niet op ondergrens/bovengrens voor waarde complete oplossing
- **Bt** kán besluiten om deeloplossing niet verder uit te breiden als ondergrens/bovengrens voor waarde complete oplossing niet beter is dan beste waarde van tot nu toe gevonden complete oplossing, maar dit hoeft niet
- **B&B** breidt deeloplossing niet verder uit als ondergrens/bovengrens voor waarde complete oplossing niet beter is dan beste waarde van tot nu toe gevonden complete oplossing
- **ES** houdt op elk moment slechts één deeloplossing bij (en impliciet de daaraan voorafgaande deeloplossingen in de toestandsboom)
- **Bt** houdt op elk moment slechts één deeloplossing bij (en impliciet de daaraan voorafgaande deeloplossingen in de toestandsboom)
- **B&B** houdt verzameling van deeloplossingen bij die (mogelijk) nog uitgebreid worden tot complete oplossing

- **ES** genereert bij uitbreiden van deeloplossing één kind tegelijk
Bt genereert bij uitbreiden van deeloplossing in principe één kind tegelijk
B&B genereert bij uitbreiden van deeloplossing meteen alle kinderen
- **Bt** kan ondergrens/bovengrens gebruiken om deeloplossingen te selecteren die als eerstvolgende worden uitgebreid, maar alleen om te kiezen tussen de kinderen van de huidige deeloplossing, en bovendien is dit niet standaard
B&B gebruikt ondergrens/bovengrens om deeloplossing te selecteren die als eerstvolgende wordt uitgebreid
- **ES** volgt (in de praktijk) depth-first wandeling in toestandsboom bij zoeken naar (optimale) oplossing
Bt volgt depth-first wandeling in toestandsboom bij zoeken naar (optimale) oplossing
B&B kan heen en weer springen in toestandsboom bij zoeken naar optimale oplossing

Als je gewoon wilt doortekenen in één plaatje... (alternatief)

A	B	C	D	E	F	G	H	Actie
0	∞	∞	∞	∞	∞	∞	∞	Begin met A
-	2	9	∞	∞	∞	∞	∞	Kies B, vanaf A
-	-	9	6	∞	4	∞	∞	Kies F, vanaf B
-	-	9	6	∞	-	∞	11	Kies D, vanaf B
-	-	8	-	∞	-	∞	11	Kies C, vanaf D
-	-	-	-	12	-	10	11	Kies G, vanaf C
-	-	-	-	12	-	-	11	Kies H, vanaf F
-	-	-	-	12	-	-	-	Kies E, vanaf C



Een rij in de tabel komt overeen met het array pad in pseudo-code op volgende slide.

// invoer: **samenhangende** gewogen graaf $G = (V, E)$ en startknoop s
 // uitvoer: array dat de lengtes van de kortste paden vanuit s bevat;
 // na afloop is $\text{pad}[v] =$ de lengte van een kortste pad van s naar v

```

for  $v \in V$  do
     $\text{pad}[v] := \infty$ ;
od
 $\text{pad}[s] := 0$ ;
 $U := \emptyset$ ;

while ( $U \neq V$ ) do
    vind knoop  $v^* \in V \setminus U$  met  $\text{pad}[v^*]$  minimaal;
     $U := U \cup \{v^*\}$ ;
    for alle knopen  $v$  aangrenzend aan  $v^*$  do
        if  $\text{pad}[v^*] + \text{gewicht}(v^*, v) < \text{pad}[v]$  then
             $\text{pad}[v] := \text{pad}[v^*] + \text{gewicht}(v^*, v)$ ;
        fi
    od
od
    
```

Complexiteit...

De volgorde waarin de knopen (deeloplossingen) worden uitgebreid hangt direct af van de berekende grenzen:

- er worden meerdere deeloplossingen tegelijk bijgehouden, dit in tegenstelling tot backtracking
- in elke stap wordt een van al deze deeloplossingen gekozen, en daarvan worden alle 1-staps-uitbreidingen (kinderen in de state space tree) bekeken en geëvalueerd (d.w.z. ondergrens/bovengrens bepaald)
- zinloze uitbreidingen worden meteen verworpen
- meestal wordt de deeloplossing gekozen die het meest veelbelovend lijkt: **best-fit-first branch-and-bound**
- bij minimalisatieproblemen (resp. maximalisatieproblemen) kiezen we de knoop met de laagste ondergrens (resp. hoogste bovengrens) als eerste

Bij Dijkstra: kies knoop buiten boom met laagste kandidaatwaarde

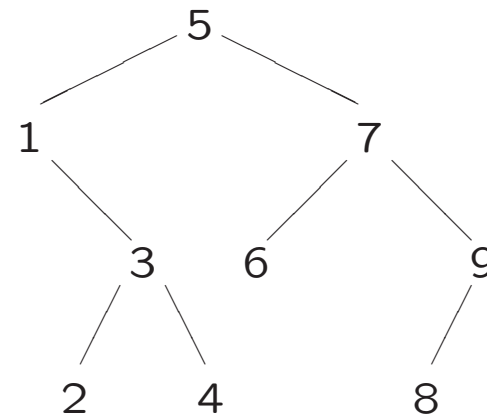
Bij Branch & Bound: kies deeloplossing met beste ondergrens/bovengrens

Priority Queue:

- objecten met prioriteit (en info)
- insert
- findmax (findmin)
- deletemax (deletemin)
- (optioneel) changepriority

Binaire zoekboom:

LWR
 1 2 3 4 5 6 7 8 9



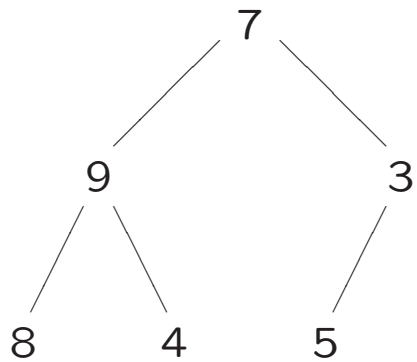
```
class knoop
{
    int info;
    knoop* links;
    knoop* rechts;
public:
    knoop ( ) // constructor
    {
        info = 0;
        links = NULL;
        rechts = NULL;
    }
}; // knoop
```

De binaire boom wordt gerepresenteerd door middel van een pointer naar de wortel:

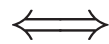
```
knoop* wortel; // de ingang tot de binaire boom
```

Netter om een klasse te gebruiken: zie [Programmeermethoden](#)

- Een **complete** binaire boom is een binaire boom waarbij alle nivo's geheel vol zitten, behalve eventueel het onderste. Op het onderste nivo mogen alleen de meest rechter knopen missen.
- Voorbeeld:

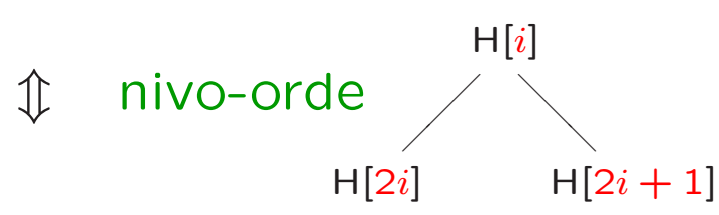
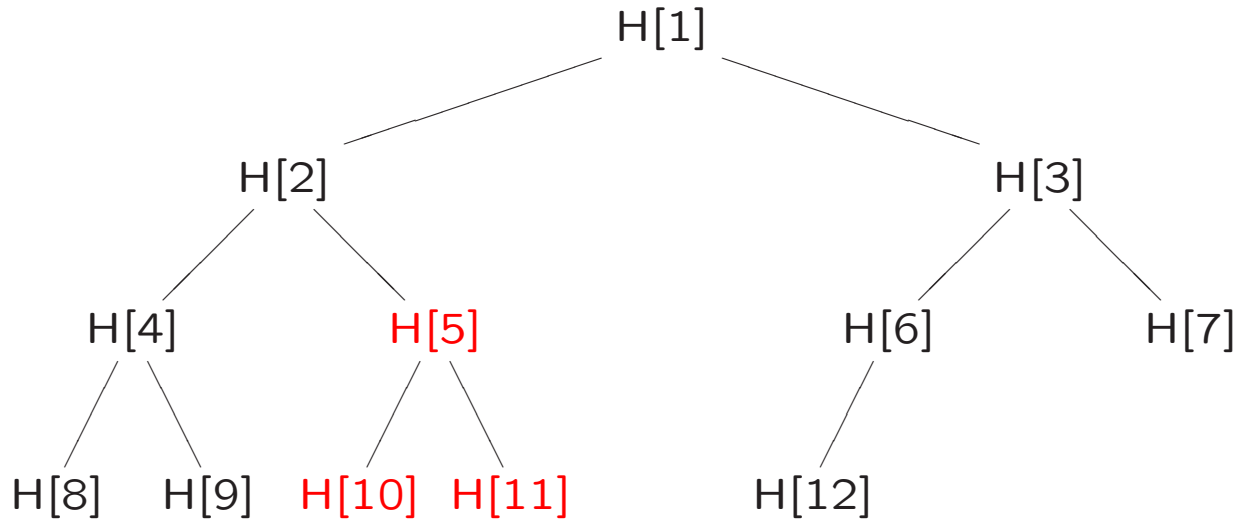


complete binaire boom



7 9 3 8 4 5

representatie als **array**



H[1] H[2] H[3] H[4] H[5] H[6] H[7] H[8] H[9] H[10] H[11] H[12]

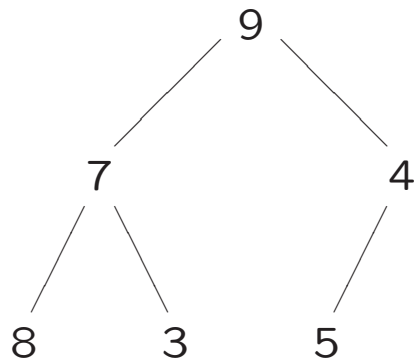
Definitie

Een **heap** (hoopstructuur) is een binaire boom met de volgende twee eigenschappen:

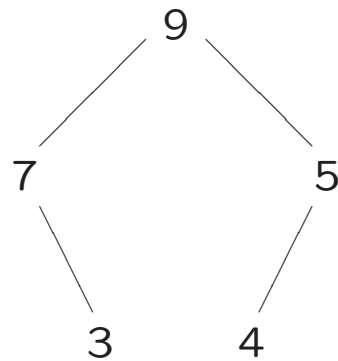
1. **Vorm:** de binaire boom is **compleet**
2. **Inhoud:** de **heap-eigenschap** geldt, d.w.z. in elke knoop geldt dat de waarde opgeslagen in die knoop **groter dan of gelijk is aan**(*) de waarde in zijn kinderen

Langs elk pad van de wortel tot een blad zijn de sleutels in de knopen dus van groot naar klein geordend.

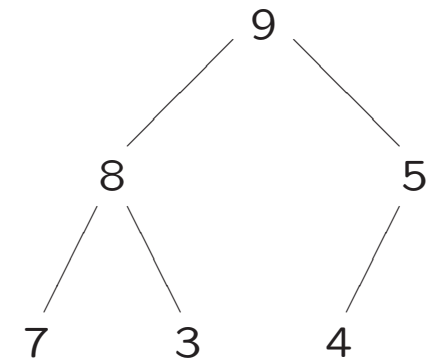
(*) we spreken dan wel van een **max-heap**; een **min-heap** wordt analoog gedefinieerd



1. geen heap



2. geen heap



3. wel heap

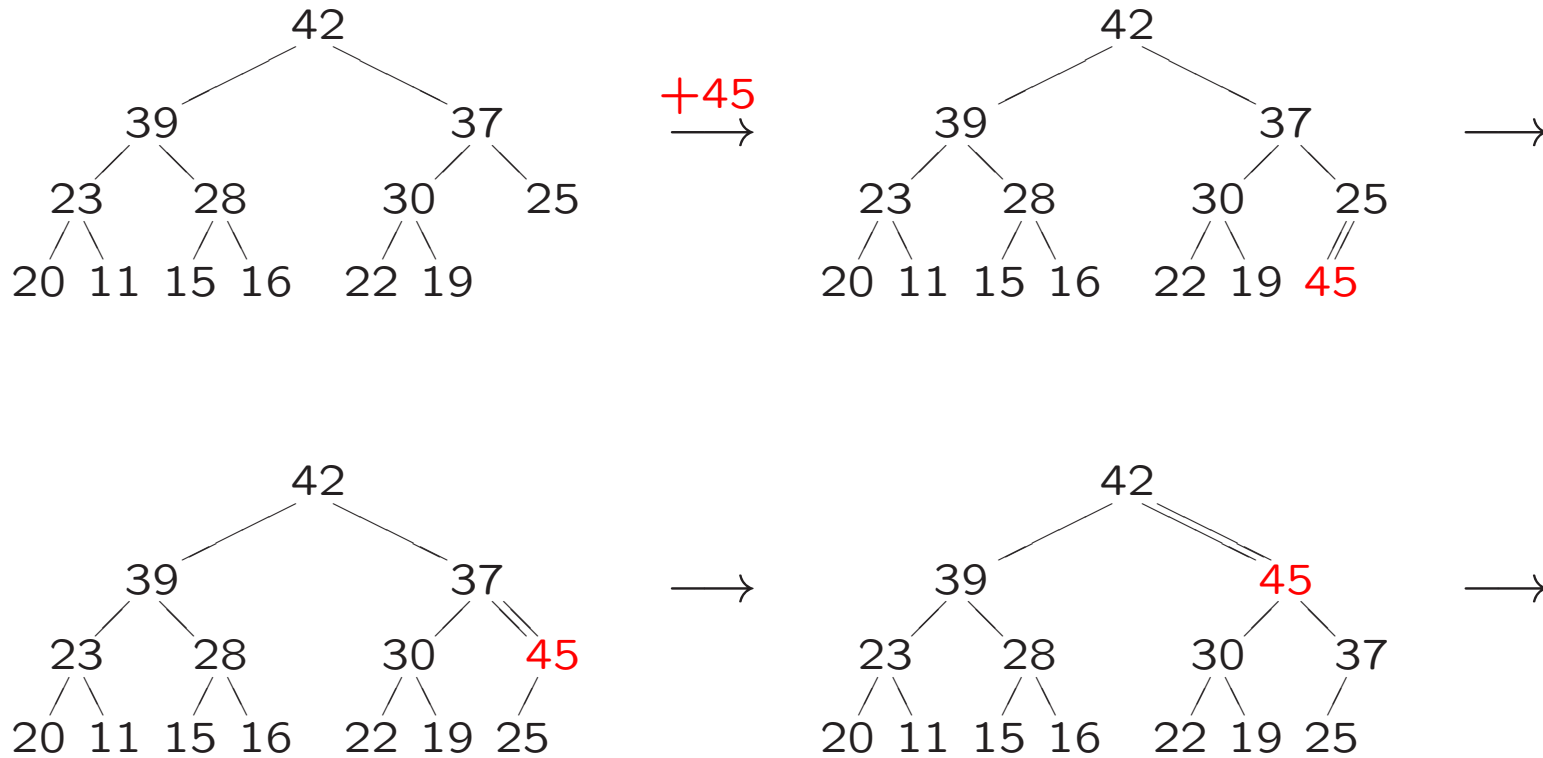
1. ouder \geq kinderen geldt niet in elke knoop
2. niet compleet
3. compleet en ouder \geq kinderen

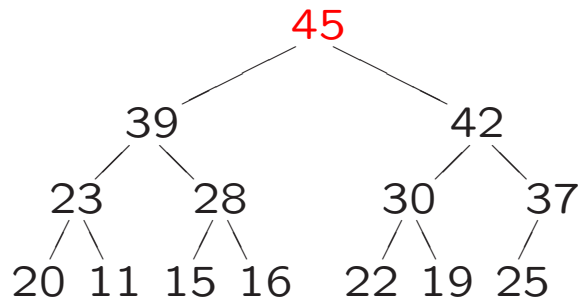
1. Gegeven n , dan bestaat er precies één **complete** binaire boom met n knopen. Deze heeft hoogte $\lfloor \lg n \rfloor$.
2. De wortel van een heap bevat altijd de grootste waarde.
3. Voor elke knoop van een heap geldt: de subboom met die knoop als wortel is weer een heap.
4. Een heap wordt gerepresenteerd door een **eendimensionaal array** H , met de inhoud van de n knopen op posities 1 t/m n .
 - ouderknopen (interne knopen) corresponderen met de posities 1 t/m $\lfloor \frac{n}{2} \rfloor$; bladeren met $\lfloor \frac{n}{2} \rfloor + 1$ t/m n .
 - de kinderen van $H[i]$ ($i = 1, \dots, \lfloor \frac{n}{2} \rfloor$) zijn $H[2i]$ en $H[2i + 1]$; de ouder van $H[i]$ ($i = 2, \dots, n$) is $H[\lfloor i/2 \rfloor]$.

Wanneer de waarde in een knoop verandert (of een waarde wordt verwijderd/toegevoegd) zal i.h.a. de heap-eigenschap niet meer gelden. Er zijn twee manieren (beide $O(\lg n)$, met n het aantal knopen van de heap) om die weer te herstellen, afhankelijk van de situatie.

Stel nu dat de waarde in één knoop veranderd wordt. Dan zijn er twee mogelijkheden:

1. waarde in knoop $>$ waarde in ouder: herhaald verwisselen met ouder totdat de heap-eigenschap hersteld is (waarde **borrelt omhoog**)
2. waarde knoop $<$ waarde van (ten minste een der) kinderen: herhaald verwisselen met grootste kind totdat de heap-eigenschap hersteld is (waarde **zakt omlaag**)





Heap-eigenschap weer hersteld

42 39 37 23 28 30 25 20 11 15 16 22 19

⇓

42 39 37 23 28 30 25 20 11 15 16 22 19 45

⇓

42 39 37 23 28 30 45 20 11 15 16 22 19 25

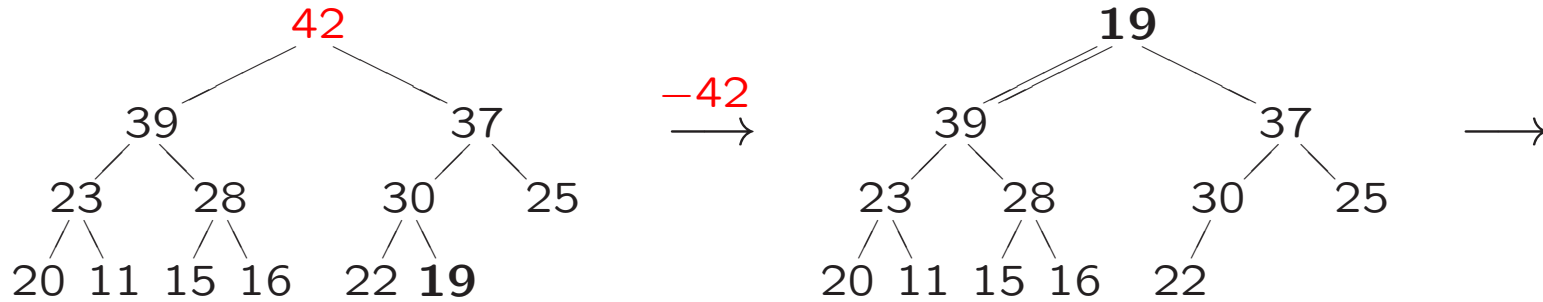
⇓

42 39 45 23 28 30 37 20 11 15 16 22 19 25

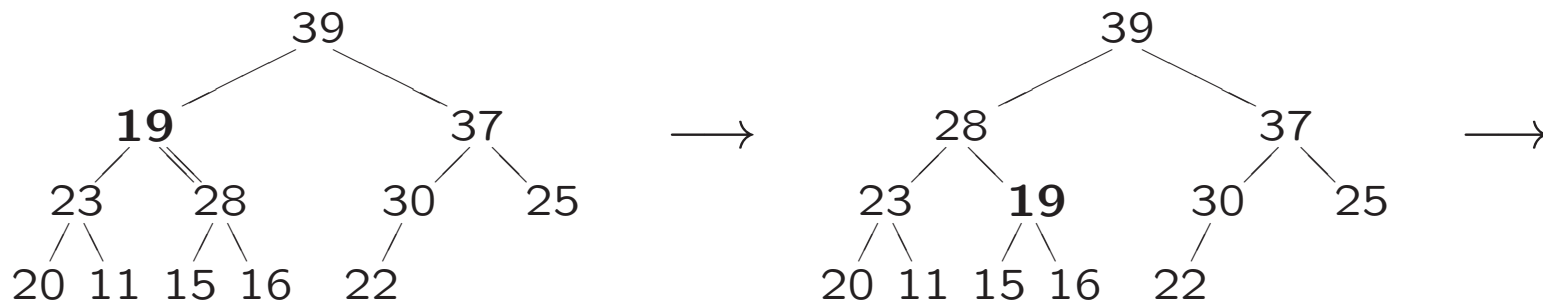
⇓

45 39 42 23 28 30 37 20 11 15 16 22 19 25

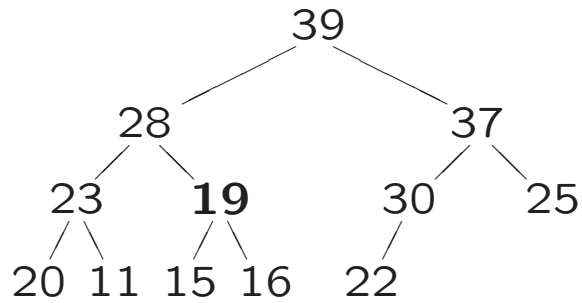
-1-



links heap, rechts heap



-2-



Heap-eigenschap weer hersteld

42 39 37 23 28 30 25 20 11 15 16 22 19

⇓

19 39 37 23 28 30 25 20 11 15 16 22

⇓

39 19 37 23 28 30 25 20 11 15 16 22

⇓

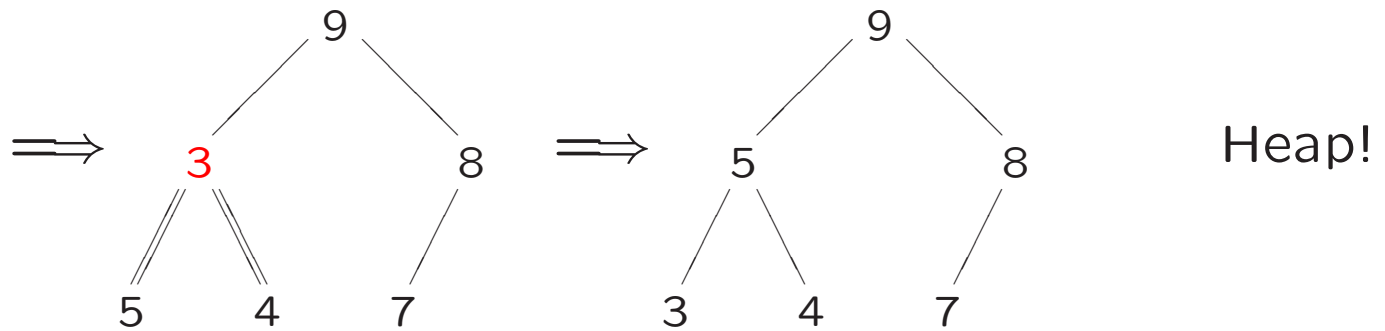
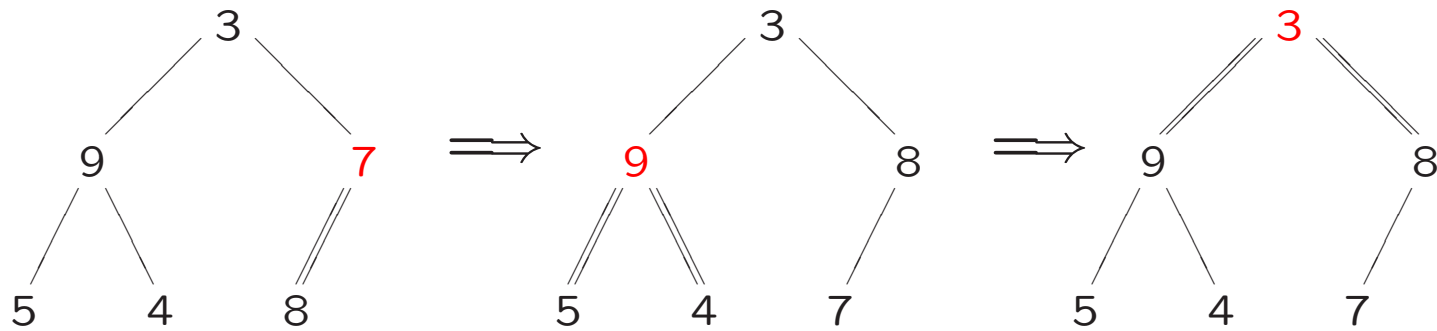
39 28 37 23 19 30 25 20 11 15 16 22

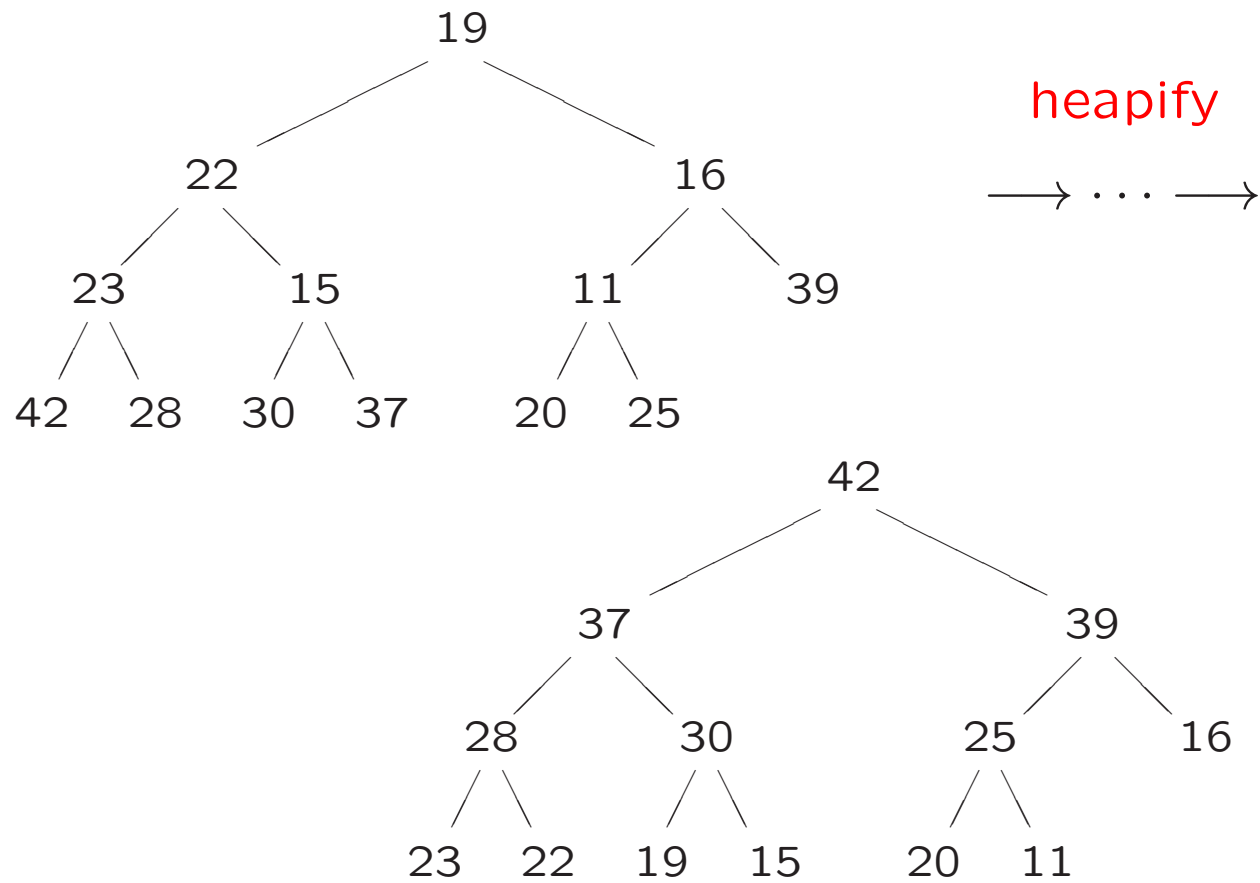
⇓

39 28 37 23 19 30 25 20 11 15 16 22

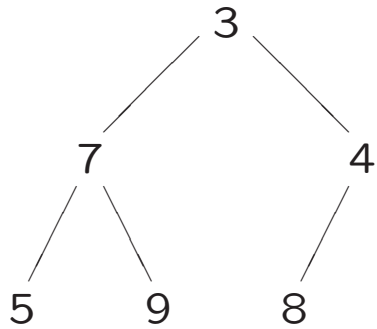
Constructie van een heap uit een gegeven rij (array H) sleutels (getallen bijv.):

- **Bottom up (heapify)**: beginnend bij $H[\lfloor \frac{n}{2} \rfloor]$, de laatste ouderknoop, en zo teruglopend, wordt in alle subbomen met de ouderknopen als wortel de heap-eigenschap hersteld via omlaag zakken van de (inhoud van die) ouder-knoop
- **Top down**: beginnend bij de wortel en door telkens een knoop meer bij de heap te betrekken wordt de heap-eigenschap steeds hersteld via omhoog borrelen van de (inhoud van de) nieuwe knoop





```
for  $i := \lfloor \frac{n}{2} \rfloor$  downto 1 do  
     $k := i; v := H[k];$   
    heap := false;  
    while not heap and  $2 * k \leq n$  do // geen blad  
         $j := 2 * k;$  // linkerkind  
        if  $j < n$  then // 2 kinderen  
            if  $H[j] < H[j + 1]$  then  
                 $j := j + 1;$  fi // selecteer grootste kind  
            fi  
        if  $v \geq H[j]$  then  
            heap := true;  
        else  
             $H[k] := H[j]; k := j;$  fi  
    od  
     $H[k] := v;$   
od
```



- Naam bottom-up – top-down
- Worst case bottom-up (heapify)...
Worst case complexiteit...
- Worst case top-down...
Worst case complexiteit...

- Naam bottom-up – top-down
- Worst case bottom-up (heapify)...

Worst case complexiteit: $O(n)$

- Worst case top-down...

Worst case complexiteit: $O(n \lg n)$

<http://oeis.org/A036799>

1. Maak een heap van het gegeven array
2. Verwijder nu herhaald ($n - 1$ keer) de grootste waarde uit de wortel:
 - verwissel deze met de laatste waarde uit de heap
 - verlaag de grootte van de heap met 1
 - zorg dat overal de heap-eigenschap weer geldt door de nieuwe waarde uit de wortel te laten zakken

Het array wordt zo oplopend gesorteerd.

Complexiteit: $O(n \lg n)$.

Sorteer het array 3 9 7 5 4 8 met heapsort.

Fase 1

3 9 7 5 4 8 →
 3 9 8 5 4 7 →
 9 3 8 5 4 7 →
 9 5 8 3 4 7

Fase 2

9 5 8 3 4 7 →
 7 5 8 3 4 |9 →
 8 5 7 3 4 |9 →
 4 5 7 3 |8 |9 →
 7 5 4 3 |8 |9 →
 3 5 4 |7 |8 |9 →
 5 3 4 |7 |8 |9 →
 4 3 |5 |7 |8 |9 →
 3 |4 |5 |7 |8 |9

- **Lezen/leren bij dit college:**

slides, paragraaf 12.2, paragraaf 6.4

- **Werkcollege:**

donderdag 19 mei 2016, 13:45–15:30, in zaal B01/B02

- **Opgaven:**

zie <http://www.liacs.leidenuniv.nl/~graafjmde/ALGO>

- **Vanmiddag 15.30–17.30:** assistentie opdracht 3

- **Volgend ‘college’:**

vrijdag 20 mei 2016: tentamen oefenen