

datastructuren en algoritmen  
2011 09 12  
college 3

- **lineaire datastructuren**  
stacks, queues, vectoren, lijsten, rijtjes
- **hierarchische datastructuren**  
bomen, traversals

## schema

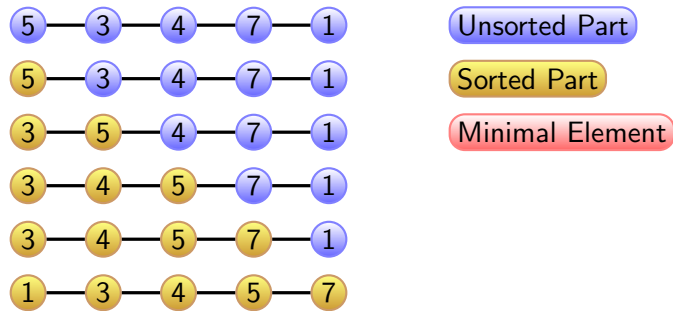
- recap
- sorteren
- heaps
- materiaal

## schema

- recap
- sorteren
- heaps
- materiaal

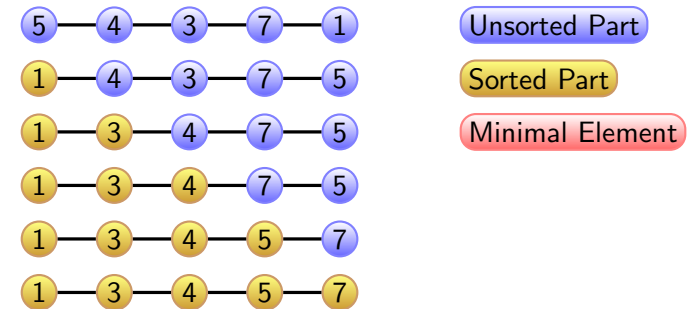
## sorteren: voorbeeld insertion sort

- het rijtje bestaat uit een gesorteerd deel en een niet-gesorteerd deel
- neem het eerste item uit het niet-gesorteerde deel en voeg het op de juiste plek toe aan het gesorteerde deel



## sorteren: voorbeeld selection sort

- zoek een kleinste key in het rijtje en verwissel dat item met het eerste item
- itereer op de rest van het rijtje



## sorteren: specificatie

- **input:**  
een eindig rijtje getallen
- **output:**  
een geordende permutatie van de input

## wat sorteren we?

we sorteren alleen getallen, maar:

- zo'n getal is de **key** van een groter **item**  
een **item** bestaat uit een **key** en **element**
- (we maken ons niet druk over bijv de vraag of we hele grote items moeten verplaatsen)

## wat is gesorteerd?

### definitie:

een **ordering** is een binaire relatie  $\leq$  zo dat

- $n \leq n$   
**reflexiviteit**
- als  $m \leq n$  en  $n \leq p$  dan  $m \leq p$   
**transitiviteit**
- als  $m \leq n$  en  $n \leq m$  dan  $m = n$   
**anti-symmetrie**

een ordening is **totaal** als elk tweetal elementen vergelijkbaar is

**voorbeeld & meestal gebruikt:** natuurlijke getalen met  $\leq$

## eigenschappen van sorteeralgoritmes

- **de worst-case tijdscomplexiteit**  
in termen van grote-Oh
- **in place**  
ruimtegebruik in  $\mathcal{O}(1)$ : dat voor de data  
plus eventueel een constante hoeveelheid extra ruimte
- **stabiel**  
de volgorde van items met dezelfde key blijft behouden

## hoe sorteren we?

- er zijn veel **sorteeralgoritmes**  
(we gaan ook heel vaak sorteren!)
- veel hebben als basis vergelijken van keys:  
**comparison sort**

## sorteren: conventies

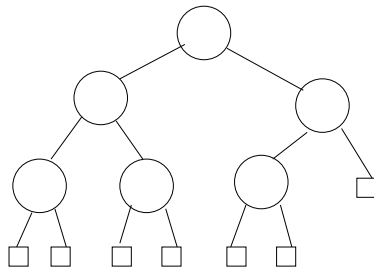
- we gaan uit van een **totale ordening** op de keys
- meestal zijn we alleen geïnteresseerd in de **keys**  
(en niet zozeer in de elementen)
- een key kan **meer dan eens** voorkomen
- we bestuderen **verschillende sorteeralgoritmes**

## heaps

een data-structuur die gebruikt wordt voor sorteren  
maar ook voor andere toepassingen

### complete binaire boom: definitie

- de lagen  $0, 1, \dots, \text{hoogte} - 2$  zijn helemaal vol met interne knopen  
dus  $2^i$  interne knopen in zo'n laag  $i$
- de laag  $\text{hoogte} - 1$  is vanaf links vol met interne knopen,  
dan leeg

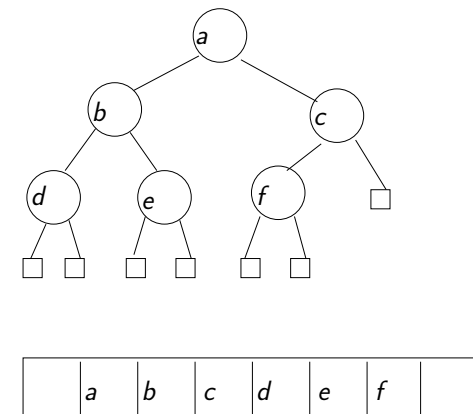


## heap: definitie

- een complete binaire boom
- keys op de interne knopen
- externe knopen zijn leeg
- met de (min- of max-)heap eigenschap

### heap als vector

met de level-numbering  
werkt mooi voor complete binaire bomen!  
heap met  $n$  items als array met lengte  $m \geq n$



## toegang tot keys in de heap

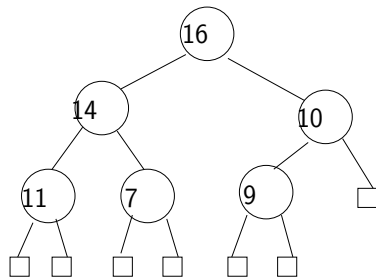
vanuit de boom gezien geven de we array-index:

- $v$  is wortel dan  $p(v) = 1$
- $v$  linker-kind van  $u$  dan  $p(v) = 2p(u)$
- $v$  rechter-kind van  $u$  dan  $p(v) = 2p(u) + 1$

vanuit het array gezien geven we ouder of kind:

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{leftChild}(i) = 2i$
- $\text{rightChild}(i) = 2i + 1$

## max-heap: voorbeeld

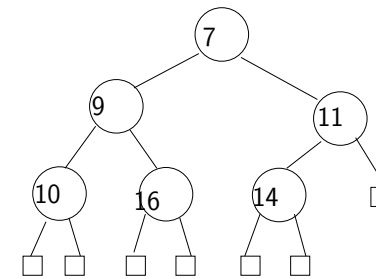


	16	14	10	11	7	9	
--	----	----	----	----	---	---	--

## heap-eigenschap: definities

- voor de min-heap:  
als boom: key van ouder kleiner-gelijk key van kind  
als array:  $H[\text{parent}(i)] \leq H[i]$   
naar beneden worden de keys niet echt kleiner  
minimum key op de wortel
- voor de max-heap:  
als boom: key van ouder groter-gelijk key van kind  
als array:  $H[\text{parent}(i)] \geq H[i]$   
naar beneden worden de keys niet echt groter  
maximum key op de wortel

## min-heap: voorbeeld



	7	9	11	10	16	14	
--	---	---	----	----	----	----	--

## heap: eigenschap

de hoogte van een heap met  $n$  interne knopen is  $\lceil \log(n+1) \rceil$

dus in  $\Theta(\log(n))$

## down-heap bubbel

- voor max-heap of voor min-heap
- heap gezien als boom of als array
- gebruikt bij verwijderen en bij sorteren

## bubbel in heaps

- we hebben een knoop met links en rechts een al een heap  
herstel heap-eigenschap door down-heap bubbel
- we hebben een heap en voegen een item toe  
herstel heap-eigenschap door up-heap bubbel

## down-heap bubbel voor de min-heap

$\text{downMinHeap}(H, i)$  met input  $i$  een knoop in  $H$   
links en rechts van  $i$  is het al een min-heap

- bekijk  $i$ , zijn linker-kind  $l$  en zijn rechter-kind  $r$
- bepaal de kleinste van  $i, l, r$
- als kleinste is  $i$  dan klaar
- als kleinste is  $l$ : verwissel  $i$  en  $l$ , doe  $\text{downMinHeap}(H, i)$
- als kleinste is  $r$ : verwissel  $i$  en  $r$ , doe  $\text{downMinHeap}(H, i)$

## up-heap bubbel

- voor max-heap of voor min-heap
- heap gezien als boom of als array
- gebruikt bij toevoegen

## tijdscomplexiteit bubbel

voor down-heap bubbel en up-heap bubbel geldt:  
tijdscomplexiteit in  $\mathcal{O}(\text{hoogte})$  dus in  $\mathcal{O}(\log(n))$

## up-heap bubbel voor de min-heap

$\text{upMinHeap}(H, i)$  met input  $i$  een knoop in  $H$

$i$  verstoort mogelijk de heap-eigenschap

- bekijk  $i$  en zijn ouder  $p$
- als  $i$  kleiner: verwissel  $i$  en  $p$ , doe  $\text{upMinHeap}(H, i)$
- als  $i$  groter: klaar

## heap: verwijder de wortel

- bewaar wortel; zet laatste op plek wortel  
in array: verwissel  $H[1]$  met  $H[\text{laatste}]$ ; dan  $\text{laatste} := \text{laatste} - 1$
- zorg dat het weer een heap wordt met  $\text{downMinHeap}$  of  $\text{downMaxHeap}$
- worst-case tijdscomplexiteit:  $\mathcal{O}(\log(n))$  (waarom?)

## heap: toevoegen

input: een heap  $H$  en een item met key  $k$

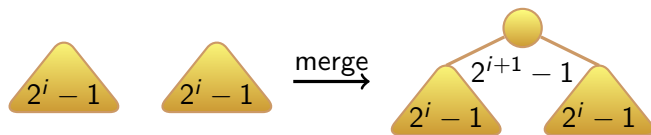
- plaats item op plek *laatste*; hoog *laatste* een op
- zorg dat het weer een heap wordt met upMinHeap of upMaxHeap

## hoe maken we een heap van $n$ items?

- voeg één voor één een item toe  
per stap in  $\mathcal{O}(\log(n))$  dus totaal in  $\mathcal{O}(n \log n)$
- voor  $n = 2^h - 1$  voor zekere  $h$ : bottom-up heap construction  
in  $\mathcal{O}(n)$

## bottom-up heap construction

- maak van alle  $2^h - 1$  elementen een heap ter grootte 1
- voor stap  $1, \dots, \log n$ :  
merge heaps ter grootte  $2^i - 1$  tot heap ter grootte  $2^{i+1} - 1$



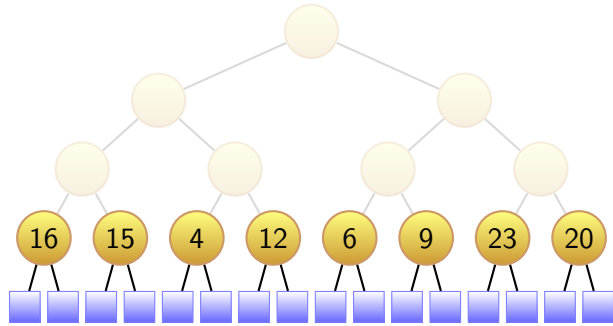
## merging twee heaps

- plak item met key  $k$  boven twee heaps
- doe zonodig down-heap bubbel om weer een heap te maken

## bottom-up heap constructie: voorbeeld

We bouwen een heap van de volgende  $2^4 - 1 = 15$  elementen:

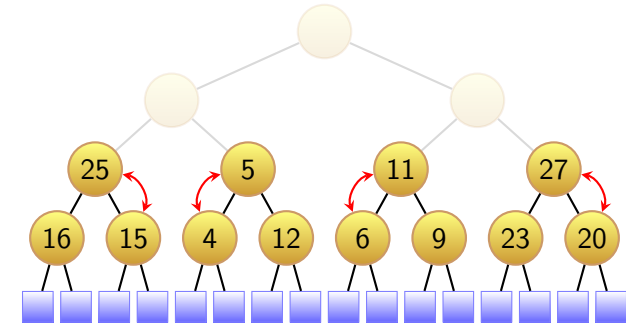
16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



## bottom-up heap construction

We bouwen een heap van de volgende  $2^4 - 1 = 15$  elementen:

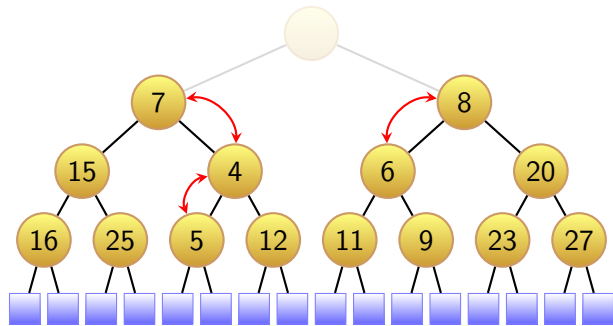
16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



## bottom-up heap construction

We bouwen een heap van de volgende  $2^4 - 1 = 15$  elementen:

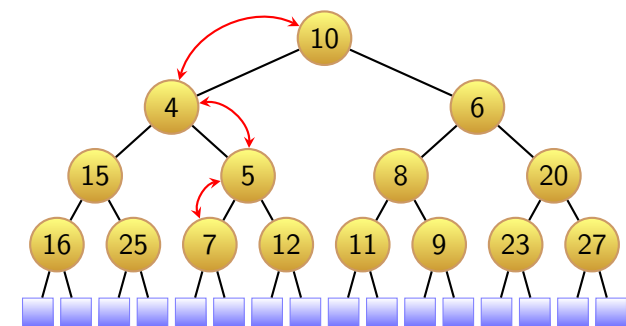
16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



## bottom-up heap construction

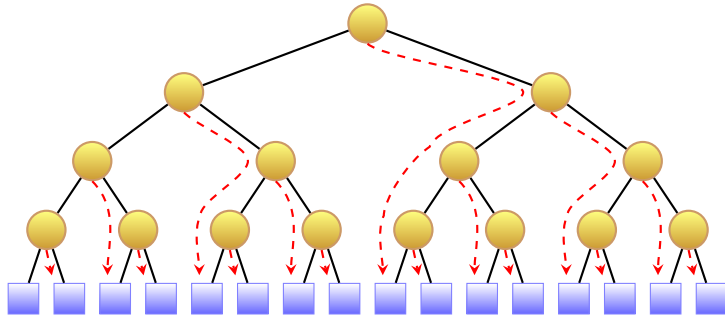
We bouwen een heap van de volgende  $2^4 - 1 = 15$  elementen:

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10



## bottom-up heap constructie: complexiteit

- we gaan ten hoogste een keer door een kant heen
- er zijn  $2n$  kanten
- totaal dus in  $\mathcal{O}(n)$  dus beter dan in  $\mathcal{O}(n \log n)$



## sorteren: heap-sort

we bekijken de max-heap gerepresenteerd als volle vector

```
Algorithm heapsort( $H$ ):  
  buildMaxHeap( $H$ )  
  for  $i = H.length$  downto 2 do  
    exchange  $H[1]$  met  $H[i]$   
     $H.size := H.size - 1$   
    downHeap( $(H, 1)$ )
```

## heap-sort: eigenschappen

- worst-case tijdscomplexiteit in  $\mathcal{O}(n \log(n))$
- in place

## schema

- recap
- sorteren
- heaps
  - bubbel
  - verwijderen en toevoegen
  - bouwen van een heap
  - sorteren met een heap
- materiaal

## materiaal

- boek 2.4.2, 2.4.3, 2.4.4

## extra materiaal

- Fibonacci heap