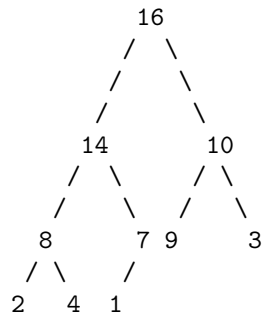




Uitwerkingen

1. Teken de max-heap $[-, 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ als boom.



2. De array-representatie:

$[-, 2, 3, 3, 4, 5, 4, 4, 6, 7, 8, 9, 5]$

3. Ten minste $2^0 + 2^1 + \dots + 2^{h-2} + 1 = 2^{h-1}$ interne knopen.
Ten hoogste $2^0 + 2^1 + \dots + 2^{h-2} + 2^{h-1} = 2^h - 1$ interne knopen.
(Zie eventueel Theorem 1.12 uit het boek.)
- 4.
5. Voor de sub-boom geldt ook de max-heap eigenschap. Dus: naar beneden lopende worden de keys niet groter. Dat geldt voor elk pad van de wortel van de sub-boom naar beneden. Die paden bevatten dus alleen keys die ten hoogste zo groot zijn als de wortel van de sub-boom.
6. Omdat alle keys verschillend zijn: op een interne knoop die alleen externe kinderen heeft. Dat is dus of op de onderste laag met interne knopen, of rechts op de een-na-onderste laag.
7. Hoeveel elementen van een heap moeten minimaal bezocht worden om het grootste element te bepalen?

We zouden natuurlijk alle elementen van de heap kunnen bekijken. Maar het kan handiger: Het grootste element van een heap bevindt zich tussen de interne knopen 'onderaan', die alleen externe opvolgers hebben. Bij een heap van hoogte h moeten we dus 2^h knopen bezoeken om het maximum te bepalen.

8. Ja.
9. Nee, door de tak van 6 naar 7. Met bijvoorbeeld 4 in plaats van 7 zou het wel een max-heap zijn.

10. In de array-representatie:

```

[-, 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]
→
[-, 27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0]
→
[-, 27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0]

```

11. Voor een max-heap: We gaan uit van een input bestaande uit een bijna-heap A in array-representatie, en een index i in dat array. De methoden `leftChild` en `rightChild` berekenen de indexen in de array-representatie van linker- en rechter-kind.

Algorithm `downMaxHeap(A, i)`:

```

l := leftChild(i)
r := rightChild(i)
if  $l \leq A.heapsize$  and  $A[l] > A[i]$  then
    largest := l
else
    largest := i
if  $r \leq A.heapsize$  and  $A[r] > A[largest]$  then
    largest := r
if largest  $\neq i$  then
    exchange  $A[i]$  with  $A[largest]$ 
    downMaxHeap(A, largest)

```

Voor `downMinHeap`: analoog, maar $<$ in plaats van $>$, etc.

12. Drie verschillende heaps met de getallen 1, 2, 3, 4, 5, 6, 7, in vector-representatie:

[1, 2, 5, 3, 4, 6, 7]

[1, 2, 3, 4, 5, 6, 7]

[1, 2, 3, 4, 6, 5, 7]

13. (Dit is ongeveer opgave R-2.14 uit het boek.)

- Een heap waarvoor een inoder traversal het rijtje [1, 2, 3, 4, 5, 6, 7] levert bestaat niet (dat zou, in vector-representatie, de heap [4, 2, 6, 1, 3, 5, 7] zijn, maar dat is geen heap).

- Een heap waarvoor een preorder traversal het rijtje [1, 2, 3, 4, 5, 6, 7] levert bestaat wel, namelijk [1, 2, 5, 3, 4, 6, 7] (de eerste van het antwoord op de vorige opgave).
- Een heap waarvoor een postorder traversal het rijtje [1, 2, 3, 4, 5, 6, 7] levert bestaat niet; de root van de heap moet 1 zijn en kan dus geen 7 zijn.

14. (Dit is ongeveer opgave C-2.27.)

Gegeven een complete binaire boom T en zijn laatste interne knoop v is de vraag het insertion point te vinden, oftewel de eerstvolgende externe knoop in de level ordering.

Als v de root is, dan is zijn linkerkind het insertion point. Als v een linkerkind is, dan is zijn sibling het insertion point. Het moeilijke geval is als v een rechterkind is. Zoek dan de eerste voorouder p van v die zelf een linkerkind is. Zoek vervolgens de meest linker descendant van de sibling van p . Speciaal geval: als v de meest rechter knoop is op hoogte $h - 1$, dan is het insertion point het meest linker blad van de boom.

Pseudo-code van een algoritme hiervoor:

```

Algorithm insertionPoint( $T, v$ ):
  if  $T.isRoot(v)$  then
    return  $T.leftChild(v)$ 
   $p := T.parent(v)$ 
  while  $v = T.rightChild(p)$  and  $p \neq T.root()$  do
     $v := p$ 
     $p := T.parent(p)$ 
  if  $v = T.leftChild(p)$  then
     $w := T.rightChild(p)$ 
  else /*  $p = T.root()$  */
     $w := T.leftChild(p)$ 
  while  $T.isInternal(w)$  do
     $w := T.leftChild(w)$ 
  return  $w$ 

```

De tijdscomplexiteit van dit algoritme is in $\mathcal{O}(n \log(n))$.

Opmerking: hierboven werken we met de ADT van binaire bomen; een complete binaire boom is hiervan een speciaal geval. Als je concreter werkt met de implementatie met vectoren van een binaire boom, dan is het insertion point in $\mathcal{O}(1)$ te vinden: als n de index is van v in de vector, dan heeft w index $n + 1$, het volgende level number. Dat is dus beter !

15. opgave over selection sort

Selection sort is priority-queue sort waarbij de priority queue geïmplementeerd is met een ongesorteerd rijtje. Voor een ongesorteerd rijtje geldt: toevoegen is makkelijk; zoeken en verwijderen van een item met kleinste key is moeilijk. We gebruiken de operaties uit het ADT voor sequences; dat bevat ook alle operaties uit de ADTs voor vectoren en lijsten.

Pseudo-code voor het algoritme selection sort:

```

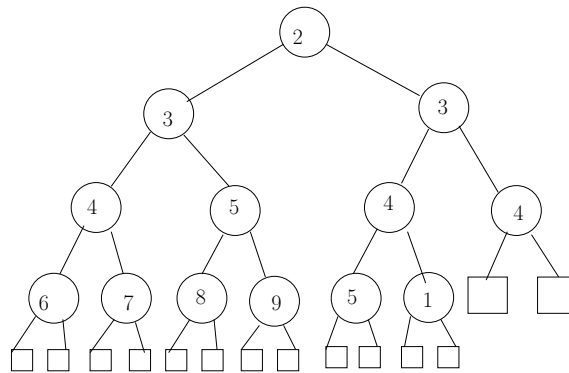
Algorithm SelectionSort( $C, U$ ):
  while not  $C.isEmpty()$  do
     $e := C.remove(first())$ 
     $U.insertItem(e, e)$ 
  while not  $U.isEmpty()$  do
     $e := U.removeMin()$ 
     $C.insertLast(e)$ 

```

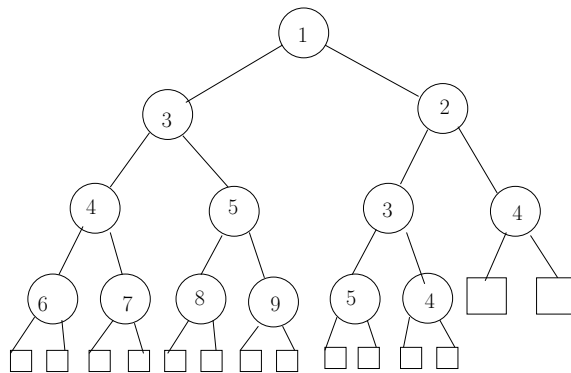
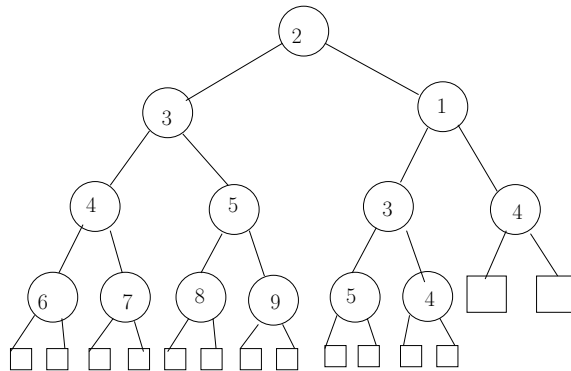
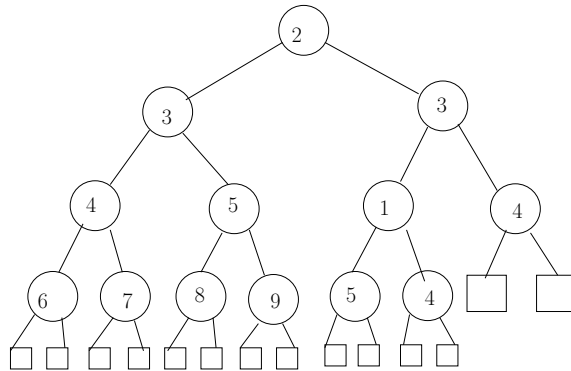
16. opgave over insertion sort

Pseudo-code voor het algoritme insertion sort is hetzelfde, behalve dat we dan een priority queue S gebruiken die geïmplementeerd is met een gesorteerd rijtje. Voor een gesorteerd rijtje geldt: toevoegen is moeilijk, maar het zoeken en verwijderen van een item met kleinste key is makkelijk.

17. We voegen een knoop met key 1 toe op het insertion point; dit levert:

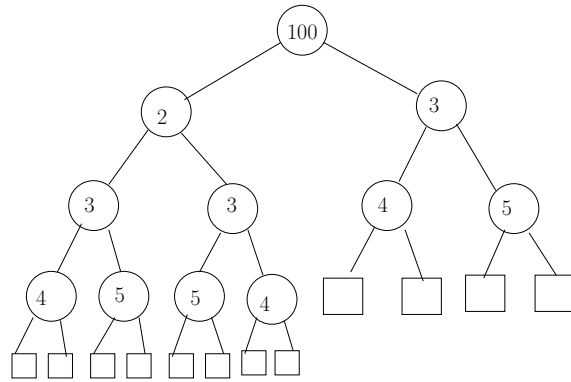


Dan volgen er drie bubbel-stappen met `upMinHeap`, als volgt:

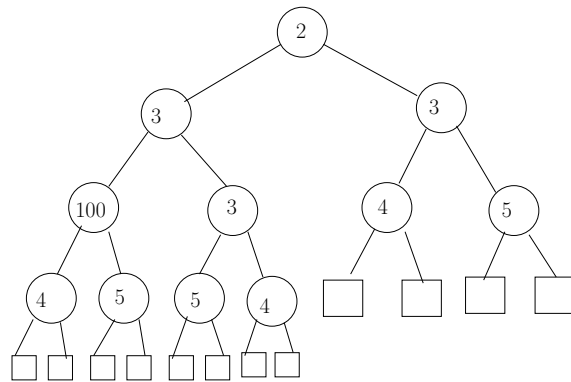
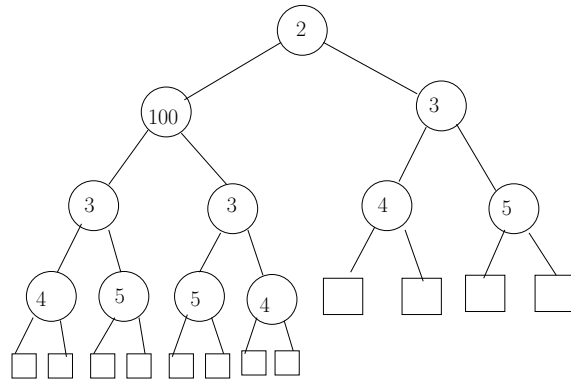


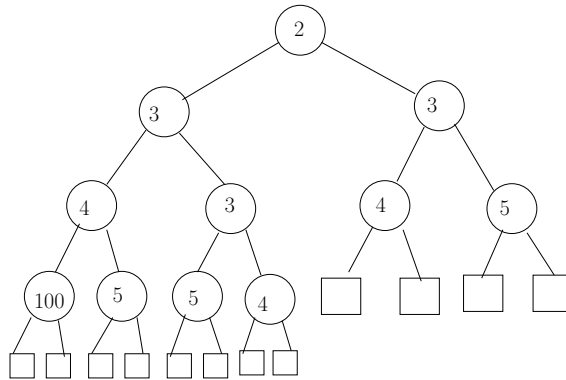
Het herstellen van de min-heap-order vergt hier maximum aantal stappen.

18. Om te beginnen verwijderen we het item uit de wortel (dit wordt bewaard om te kunnen retourneren maar dat zie je niet in het plaatje), en zetten op de wortel het item dat ‘achteraan’ staat, oftewel de meest rechter interne knoop uit laag $h - 1$. Dit levert:



Dan volgen er drie bubbel-stappen met `downMinHeap`, als volgt:



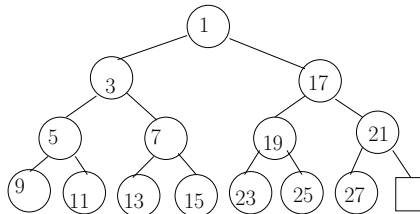


Het herstellen van de min-heap-order vergt ook hier een maximum aantal stappen.

NB: bij heaps kunnen we een willekeurig item (dus paar van key en element) toevoegen door het in eerste instantie op het insertion point te plaatsen en vervolgens zoveel up-heap bubbel-stappen uit te voeren als nodig. We verwijderen *alleen* het item dat op de wortel staat. In dat geval vervangen we het item op de wortel door het ‘laatste’ item, en vervolgens doen we zoveel down-heap bubbel-stappen als nodig.

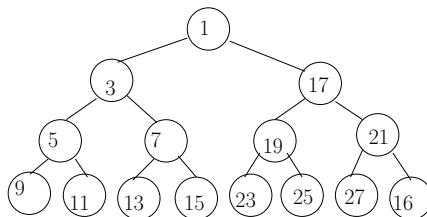
19. (Dit is ongeveer opgave R-2.18.)

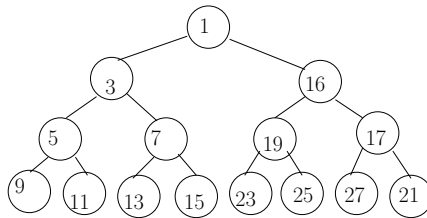
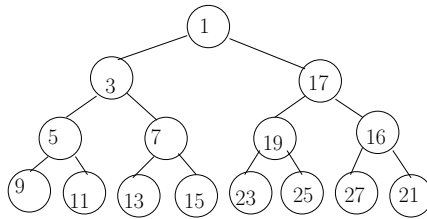
Een voorbeeld van een min-heap met als keys de eerste 14 oneven getallen, dus $1, 3, 5, \dots, 27$ zo dat voor het toevoegen van een item met key 16 twee up-heap bubbel stappen nodig zijn:



(De laatste laag met nog 14 externe knopen is hier weggelaten.) Er zijn ook andere mogelijkheden maar het gaat erom dat de getallen tot en met 15 in de linkerkant zitten.

De bubbel-stappen met up-heap:





20. Geef pseudo-code voor het algoritme `insertItem` dat als input neemt een min-heap en een item (k, e) , en dat als output geeft de heap met (k, e) eraan toegevoegd.

Algorithm `insertItem(H, k, e, , :)`
 $w := T.insertionPoint()$
zet(**k, e**)**op**node**w**
while $w.key < T.parent(w)$ **do**
 $T.swapElements(T.parent(w))$
 $w := T.parent(w)$

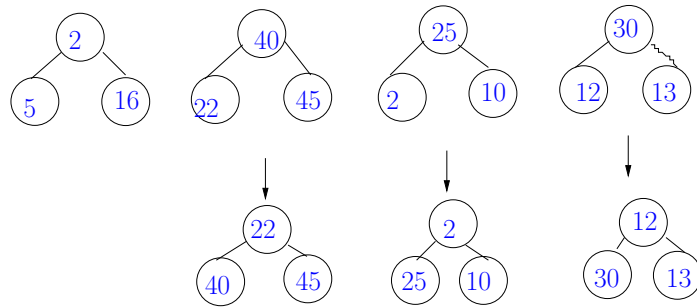
21. We passen de bottom-up heap construction toe om de volgende keys in een min-heap te plaatsen:

5, 16, 22, 45, 2, 10, 12, 13, 2, 40, 25, 30, 49, 24, 17

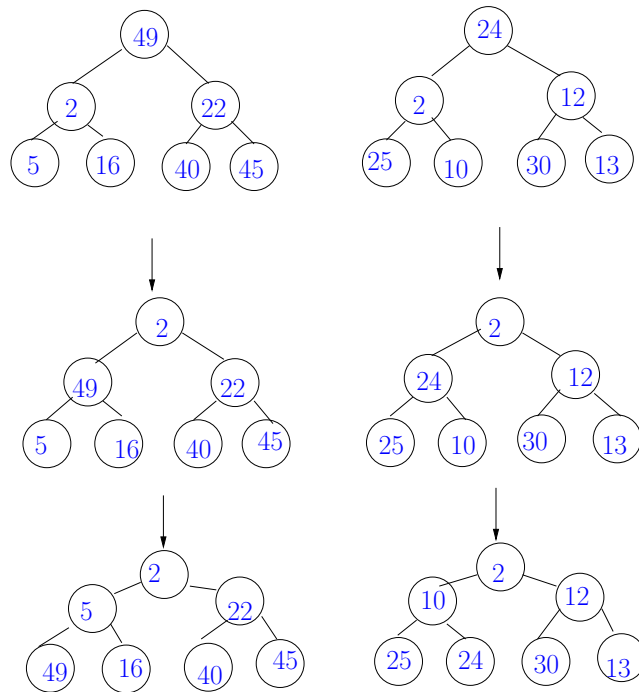
(We laten in de plaatjes de bladeren weg.) We starten met 8 kleine min-heapjes:



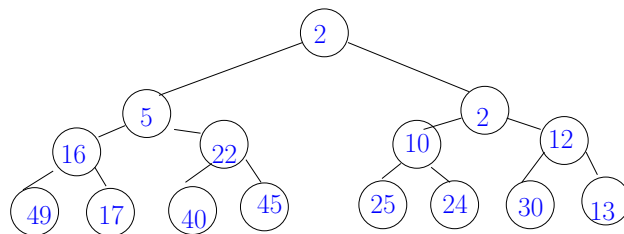
Dan voegen we twee aan twee samen met een nieuwe root, en herstellen zo nodig de heap-eigenschap:



Dan voegen we weer twee aan twee samen met een nieuwe root, en herstellen zo nodig de heap-eigenschap:



En ten slotte voegen we 17 toe, en herstellen de heap-eigenschap; resultaat:



22. Een manier: we voegen de elementen van de kleinste twee één voor één toe aan de grootste heap. Dit is in $\mathcal{O}(n \log(n + m))$ met $n \leq m$.
23. (Later.)