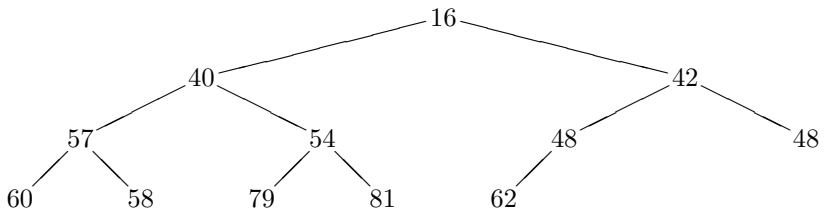
De heap

Een *complete* binaire boom is een boom waarvan alle niveaus tot en met het een-na-laagste volledig gevuld zijn en waarbij op het laagste niveau alle knopen 'zoveel mogelijk naar links' hangen.



Bovenstaande binaire boom is compleet: de eerste drie niveaus zijn volledig gevuld, en op het vierde niveau hangen alle knopen zo ver mogelijk naar links. Merk op dat het getal 48 twee keer voorkomt. Geen probleem!

We kunnen een complete binaire boom eenvoudig implementeren met behulp van een array. We slaan de knopen niveau voor niveau op in het array, en per niveau van links naar rechts. Als we het array H noemen, dan bevat $H[1]$ de wortel van de boom (we gaan er vanuit dat we de index 0 niet gebruiken), $H[2]$ bevat het linkerkind van de wortel, $H[3]$ het rechterkind van de wortel, $H[4]$ het linkerkind van het linkerkind van de wortel, enzovoort.

Het array bij bovenstaande complete binaire boom bevat achtereenvolgens de volgende getallen: 16 40 42 57 54 48 48 60 58 79 81 62.

Familie-relaties tussen de knopen kunnen simpel berekend worden uit de array indices. De kinderen van $H[i]$ zijn $H[2i]$ en $H[2i+1]$, omgekeerd is de ouder van $H[i]$ het array element $H[i \text{ div } 2]$.

Een *heap* is een complete binaire boom waarvan de knopen voldoen aan de *heap-eigenschap*. Dit betekent dat elke knoop in de boom een element bevat dat kleiner is dan (of gelijk is aan) de elementen van zijn kinderen (voorzover deze bestaan).[1] Dit betekent in het bijzonder dat het kleinste element in de wortel van de boom staat.

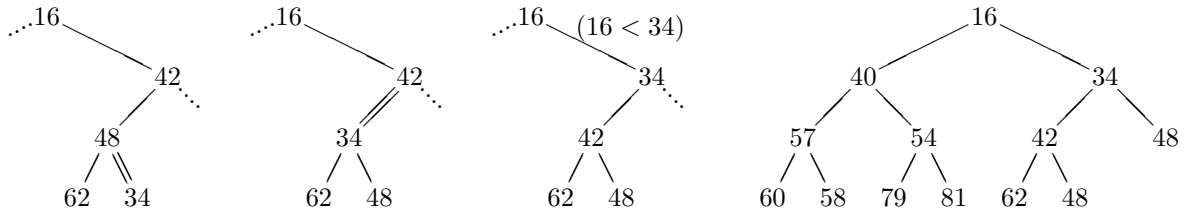
Wanneer we de waarde van een knoop veranderen zal de resulterende boom in het algemeen geen heap meer zijn. Het blijkt redelijk eenvoudig om de heap-eigenschap weer te herstellen. Is de nieuwe waarde van de knoop kleiner dan de waarde van zijn vader dan verwisselen we deze twee waardes en herhalen het proces hogerop in de boom totdat de heap-eigenschap is hersteld. De waarde *borrelt* in de boom *omhoog*. Omgekeerd, wanneer de waarde groter is dan de waarde van tenminste één van de kinderen, dan verwisselen we de waarde met de kleinste van de kinderen en herhalen dit in de richting van de bladeren totdat weer een heap is ontstaan. Deze operaties zijn vrij efficiënt, het verwisselen gebeurt op slechts één pad van de wortel naar de bladeren en kost dus maximaal een tijd die logaritmisch is in de grootte van de boom (dat wil zeggen $\mathcal{O}(\lg n)$, als de boom n elementen bevat).

We kunnen derhalve de functies BorrelOmhoog en ZinkNeer als volgt definiëren:

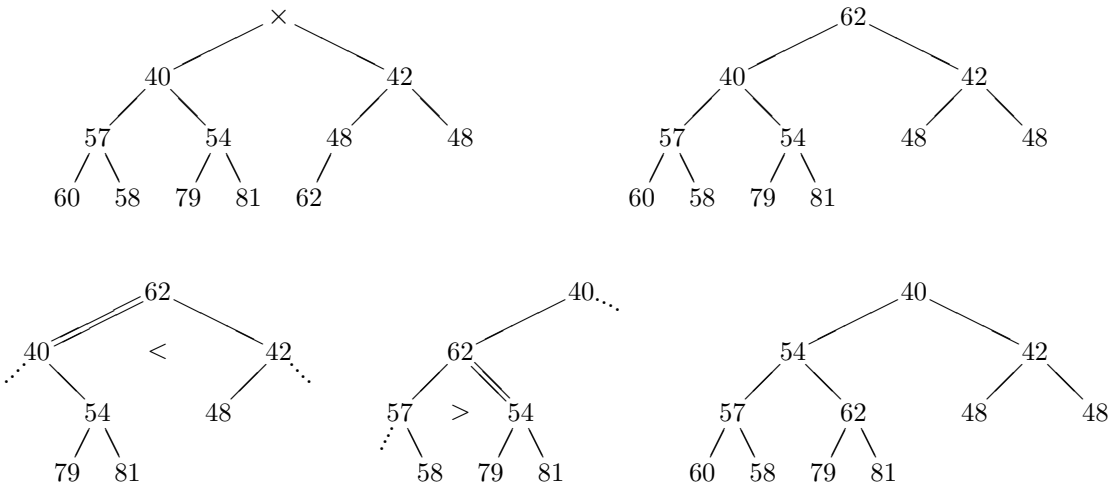
```
void BorrelOmhoog (i);
          // Zolang de waarde H[i] kleiner is dan de waarde
          // van de ouder verwisselen we deze waardes.
void ZinkNeer (i);
          // Zolang de waarde H[i] groter is dan de waarde
          // van een kind verwisselen we deze waarde met die
          // van het kleinste kind.
```

[1] In principe kan de heap-eigenschap ook net omgekeerd gedefinieerd zijn: dat elke knoop in de boom een element bevat dat groter is dan (of gelijk is aan) de elementen van zijn kinderen. Zonder beperking der algemeenheid laten we deze mogelijkheid hier echter buiten beschouwing.

Voorbeeld Het toevoegen aan en het verwijderen uit een heap. — In de voorbeeld heap voegen we 34 toe. Plaats daartoe eerst 34 in de eerste vrije positie. Borrel daarna 34 omhoog (totdat een plaats is bereikt waar de ouder kleiner is dan (of gelijk aan) 34).



In de oorspronkeljke heap verwijderen we het kleinste element (dit staat in de wortel). Om weer de eerste plek van de heap te vullen plaatsen we het laatste element in de wortel. Om de heap-eigenschap te herstellen zinken we het dan weer neer.



We kunnen de gegeven functies BorrelOmhoog en ZinkNeer ook gebruiken om van een willekeurig gevuld array een heap te maken. Dit kan bijvoorbeeld door steeds één element meer bij de heap te betrekken en deze dan omhoog te borrelen (vergelijk met toevoegen in bovenstaand voorbeeld). Dit gebeurt met de volgende pseudo-code:

```
void MaakHeapLangzaam (Grootte)
// Grootte is het aantal elementen in het array
{    int i;
     for i = 2 to Grootte
     do  BorrelOmhoog (i);
     od
}
```

Op deze manier duurt het proces $\mathcal{O}(n \cdot \lg n)$ stappen, waarbij n het aantal array elementen is. Deze waarde krijgen we door ons te realiseren dat in het slechtste geval steeds ieder element helemaal naar boven borrelt. Dit kost dan $\lfloor \lg 2 \rfloor + \lfloor \lg 3 \rfloor + \cdots + \lfloor \lg n \rfloor$ verwisselingen, wat op $\mathcal{O}(n \cdot \lg n)$ uitkomt.

Door juist aan de andere kant te beginnen krijgen we een snellere methode. We beginnen dan onderin de boom, en laten steeds elementen naar beneden zinken. Zo krijgen we allemaal kleine heapjes onderin

de boom, en die worden stap voor stap samengevoegd tot grotere heaps. Uiteindelijk is de hele boom een heap geworden. Onderstaande pseudo-code voert dit uit. Bij nauwkeurige analyse blijkt dit in lineaire tijd te werken.

```
void MaakHeap (Grootte)
// Grootte is het aantal elementen in het array
{    int i;
     for i = (Grootte div 2) downto 1
     do   ZinkNeer (i);
     od
}
```

Toepassingen Een belangrijke toepassing van de heap-structuur is voor het sorteren van elementen: we stoppen alle elementen in een array, maken een heap van dat array met (bijvoorbeeld) de functie MaakHeap, en halen één voor één de elementen eruit: eerst het kleinste element, dan het een-na-kleinste element, enzovoort.

Een andere toepassing van de heap is als representatie van de *priority queue* (*prioriteiten rij*). Bijvoorbeeld met de deadlines voor alle opdrachten van de drukkerij op blz. 91-92 van het boek van Harel. Een priority queue is een datastructuur met (o.a.) de volgende operaties: het toevoegen van een element, en het vinden en verwijderen van het *kleinste* element. In het geval van de heap kunnen we de functies BorrelOmhoog en ZinkNeer gebruiken voor het toevoegen, respectievelijk verwijderen.