Gegevensstructuren en Algoritmen

Lorenzo De Bie

June 19, 2020

Contents

1 Inleiding **3**
1.1 Analyse van algoritmen . . . 3
1.2 Asymptotische Benadering van Functies . . . 3

2 Rangschikken **4**
2.1 Insertion Sort . . . 4
2.1.1 Doel . . . 4
2.1.2 Basisprincipes . . . 4
2.1.3 Voorbeeld . . . 4
2.1.4 Complexiteit . . . 4
2.1.5 Waarom beter? . . . 4
2.1.6 Mogelijke optimalisaties? . . . 4
2.1.7 Analyse/Bewijs complexiteit . . . 5
2.2 Shell sort . . . 5
2.2.1 Doel . . . 5
2.2.2 Basisprincipes . . . 5
2.2.3 Voorbeeld . . . 5
2.2.4 Complexiteit . . . 6
2.2.5 Waarom Beter? . . . 6
2.2.6 Mogelijke Optimalisaties . . . 6
2.2.7 Analyse/Bewijs complexiteit . . . 6
2.3 Selection Sort . . . 6
2.3.1 Doel . . . 6
2.3.2 Basisprincipes . . . 6
2.3.3 Voorbeeld . . . 6
2.3.4 Complexiteit . . . 6
2.3.5 Waarom Beter? . . . 6
2.3.6 Mogelijke Optimalisaties . . . 6
2.3.7 Analyse/Bewijs complexiteit . . . 7
2.4 Heap Sort . . . 7
2.4.1 Doel . . . 7
2.4.2 Basisprincipes . . . 7
2.4.3 Voorbeeld . . . 7
2.4.4 Complexiteit . . . 10
2.4.5 Waarom Beter? . . . 10
2.5 Merge Sort . . . 10
2.5.1 Doel . . . 10
2.5.2 Basisprincipes . . . 10
2.5.3 Voorbeeld . . . 10
2.5.4 Complexiteit . . . 10
2.5.5 Waarom Beter? . . . 10
2.5.6 Mogelijke Optimalisaties . . . 11
2.5.7 Analyse/Bewijs complexiteit . . . 11
2.6 Skelet . . . 11
2.6.1 Doel . . . 11

2.6.2 Basisprincipes . . . 11
2.6.3 Voorbeeld . . . 11
2.6.4 Complexiteit . . . 11
2.6.5 Waarom Beter? . . . 11
2.6.6 Mogelijke Optimalisaties . . . 11
2.6.7 Analyse/Bewijs complexiteit . . . 11

3 Gegevensstructuren I **12**
3.1 Heaps . . . 12
3.1.1 Operaties . . . 12
3.1.2 Constructie . . . 12
3.1.3 Sterktes en Zwaktes . . . 12
3.1.4 Complexiteit van Operaties . . . 12
3.1.5 Toepassingen . . . 13
3.2 GegSkelet . . . 13
3.2.1 Operaties . . . 13
3.2.2 Sterktes en Zwaktes . . . 13
3.2.3 Complexiteit van Operaties . . . 13
3.2.4 Analyse/Bewijs Complexiteit van Operaties . . . 13
3.2.5 Toepassingen . . . 13

4 Graafalgoritmen I **14**

1 Inleiding

1.1 Analyse van algoritmen

Uitvoeringstijd staat centraal. Vergelijken van algoritmes zonder implementatie en uitvoering. Performatie afhankelijk van veel verschillende factoren:

- *programmeertaal*
- *programmeur = dummy of niet*
- *compiler*
- *processorarchitectuur*

Niet mogelijk om met al deze factoren rekening te houden && ze bepalen dan ook enkel de uitvoeringstijd voor één enkele computer. Wel rekening houden met:

- Aantal verwerkte gegevens n
- Oorspronkelijke volgorde
- Statistische verdeling

Algoritme opdelen in aantal *primitieve operaties*. Drie belangrijke situaties:

- Best-case Running Time: Ω-notatie
  Ondergrens voor uitvoertijd:
- Worst-case Running Time: O-notatie
  Bovengrens voor uitvoertijd
- Average-case Running Time: Θ-notatie
  Boven- en ondergrens voor uitvoertijd

Beschrijven stijging uitvoeringstijd ifv n

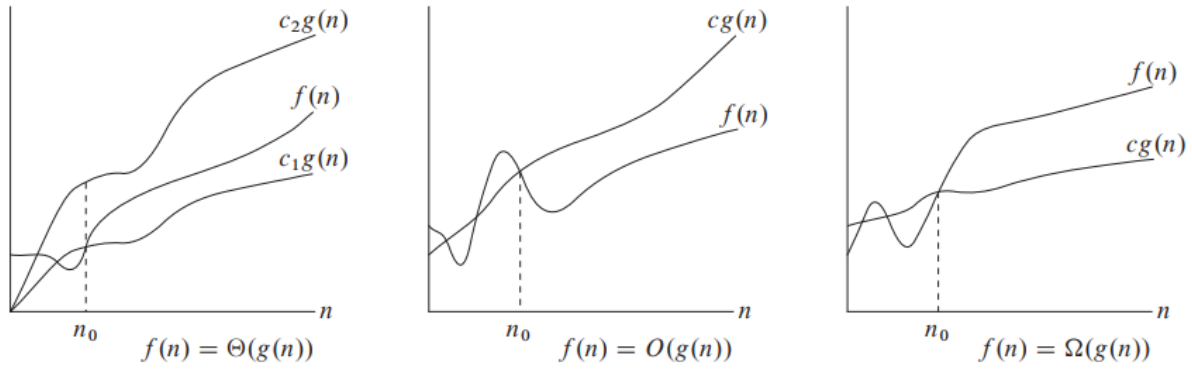


Figure 1: Voorbeelden van notaties

1.2 Asymptotische Benadering van Functies

Eigenschappen O-notatie:

- $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$
- $O(c \cdot f(n)) = O(f(n))$
- $f(n) = O(g(n))$ en $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = O(f(n))$

Per probleem asymptotische efficiëntie van efficiëntste oplossing bepalen niet altijd mogelijk (P vs NP-problemen). Sorteren wel al gekend:

Stelling 1. *Keuze tussen $f(n)$ elementen is $\Theta(\lg f(n))$ en door Stirling wordt sorteren dus $\Theta(n \lg(n))$*

Dit stuk van de cursus is echt brak so we move on

2 Rangschikken

2.1 Insertion Sort

2.1.1 Doel

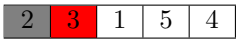
Sorteren van elementen.

2.1.2 Basisprincipes

Starten met een gesorteerde tabel van één element (is al gesorteerd). Iedere keer volgende element bijnemen en dan *tussenvoegen* op de juiste plaats door grotere elementen één voor één een plaats op te schuiven (*lineair zoeken*, beginnend bij kleinste of grootste element). Als het nieuwe element op de juiste plaats staat hebben we een gesorteerde tabel met één element meer. Dit proces wordt herhaald tot de tabel gesorteerd is. Insertion sort sorteert *ter plaatse* want het gebruikt enkel de oorspronkelijke tabel en het aantal hulpvariabelen hangt niet af van n. Stabiel sorteren is mogelijk als er strikt vergeleken wordt.

2.1.3 Voorbeeld

Rij om te sorteren: Eerste cijfer is al gesorteerd want rij van één cijfer is altijd oké. Rood = getal dat nu tussengevoegd wordt.



Geen opschuivingen nodig.

Hier moet het cijfer meerdere plaatsen opschuiven: dit gaat één plaats per keer:

Dit doen we zo verder tot de volledige rij gesorteerd is:

Het is duidelijk dat iedere keer de grotere elementen één plaats opschuiven, en het nieuwe element *tussengevoegd* wordt op de juiste plaats.

2.1.4 Complexiteit

- Best-case: $\Theta(n)$ Tabel is al gesorteerd: er zijn dus geen inversies. **Ook voor bijna gesorteerde tabellen.**
- Average-case: $\Theta(n^2)$ Iedere permutatie is even waarschijnlijk: half zoveel inversies als worst-case.
- Worst-case: $\Theta(n^2)$ Tabel staat in omgekeerde volgorde: elk paar elementen is een inversie.

2.1.5 Waarom beter?

Grote voordeel van insertion sort is dat het voor *bijna gesorteerde* tabellen ook $\Theta(n)$ is. Het werkt dus goed om elemeten toe te voegen aan een al gesorteerde tabel. Het wordt ook gebruikt in de slotfase van enkele andere (over het algemeen efficiëntere) algoritmen.

2.1.6 Mogelijke optimalisaties?

- Binair zoeken gebruiken om juiste plaats te vinden voor *tussenvoegen*. **Wel opletten dat algoritme stabiel blijft!**
- Grootste probleem is dat er **maar één inversie per verschuiving** opgelost wordt. Meerdere inversies oplossen per verschuiving zal de performantie verbeteren ⇒ *Shell sort*
- Gebruik van *swap-operatie* ipv opschuiven kan beter zijn voor grotere klassen.
- Voor begin binnenste lus checken of nieuw getal kleiner is dan kleinste (of groter dan grootste) ⇒ direct voorraan (of achteraan) plaatsen

2.1.7 Analyse/Bewijs complexiteit

De complexiteit van insertion sort hangt af van het aantal inversies in de tabel. Er zijn twee *genneste* lussen: één die over alle ementen loopt ($n-1$ keer uitgevoerd, en dus $\Theta(n)$), en de binnenste herhaling die sleutelvergelijkingen en schuifoperaties uitvoert. Iedere schuifoperatie verwijdert één inversie. Voor iedere schuifoperatie is er één sleutelvergelijking, en mogelijks nog één extra sleutelvergelijking per element. Het aantal sleutelvergelijkingen is dus $O(\#inversies + n)$

- Worst-case: elk paar elementen vormen een inversie (tabel in omgekeerde volgorde). $n(n-1)/2$ paren van elementen en dus evenveel schuifoperaties. Het aantal sleutelvergelijkingen is dus $O(n(n-1)/2 + n)$. De totale uitvoertijd is dus $\Theta(n^2)$ aangezien alle andere operaties $\Theta(n)$ zijn.
- Average-case: alle permutaties van tabel even waarschijnlijk $\Rightarrow$ helft zoveel inversies $n(n-1)/4$. Gemiddeld $\Theta(n^2)$ schuifoperaties en $O(n^2+n)$ sleutelvergelijkingen. Complexiteit blijft dezelfde: $\Theta(n^2)$.
- Best-case: geen inversies, geen kwadratische term $\Rightarrow \Theta(n)$. **Als het aantal inversies $O(n)$ blijft zal insertion sort ook $\Theta(n)$ blijven.** Bijna gesorteerde tabellen kunnen dus ook $\Theta(n)$ gesorteerd worden.

2.2 Shell sort

2.2.1 Doel

Zelfde als Insertion Sort: sorteren van elementen.

2.2.2 Basisprincipes

Shell sort is een uitbreiding op Insertion Sort. Insertion Sort is traag omdat het bij elke stap een element maar één plaats opschuift en dus maar één inversie oplost. Shell sort probeert elementen te wisselen die verder van elkaar liggen en dus meerdere inversies in één keer op te lossen. Om dit te doen wordt een tabel *'k-gerankshikt'* met steeds kleinere *'k'* (kleinste *'k'* = 1 om volledig gerangschikte tabel te garanderen). Dit algortime steunt dus op het principe dat de ordering van hogere *'k'* blijft bij het opnieuw rangschikken met kleinere *'k'*. Het bewijs hiervoor is echt te brak uitgelegd dus *good luck*. Shell sort rangschikt *ter plaatse*, maar is niet *stabiel*. Bij het sorteren van de deelreeksen kan de volgorde van gelijke sleutels wisselen.

2.2.3 Voorbeeld

54	26	93	17	77	31	44	55	20

We kunnen de tabel eerst *'k'-rangschikken* met *'k'*= 3. Om deze deeltabellen te rankshikken kan insertion sort gebruikt worden omdat de deeltabellen veel kleiner zijn. Naarmate de deeltabellen terug groter worden zijn ze wel meer gerankshikt, dus is insertion dan ook nog altijd een goede oplossing.

17	26	20	44	55	31	54	77	93

Nu kunnen we deze tabel *'2'-rangschikken*.

17	26	20	44	55	31	54	77	93

Wordt dus:

17	26	20	31	54	44	55	77	93

Als laatste *'1'-rangschikken* (wat dus gewoon neer komt op insertion sort) deze tabel. Omdat deze tabel al zo goed als gesorteerd is zal de performantie hier ook goed zijn (insertion sort is $\Theta(n)$ voor bijna gesorteerde tabellen.) Resultaat:

17	20	26	31	44	54	55	77	93

2.2.4 Complexiteit

De complexiteit van shell sort hangt af van de reeks *'k'* waarden die gebruikt worden. Gewoonlijk neemt ben een waarde kleiner dan $n/2$ als beginwaarde voor *'k'*. Er zijn veel verschillende reeksen voor *'k'*-waarden bekend, maar optimale reeks nog niet bekend.

2.2.5 Waarom Beter?

- Aanzienlijk sneller dan insertion sort, blijft ook zeer goed presteren voor grotere n.
- Eenvoudigere methode dan asymptotisch snellere algoritmen.

2.2.6 Mogelijke Optimalisaties

Enkel de reeksen voor *'k'-waarden* kan geoptimaliseerd worden.

2.2.7 Analyse/Bewijs complexiteit

Niet gegeven aangezien afhankelijk van reeks *'k'-waarden*.

2.3 Selection Sort

2.3.1 Doel

Sorteren van elementen.

2.3.2 Basisprincipes

Grootste element uit de tabel zoeken, en wisselen met het laatste element. Uit de overgebleven elementen het grootste zoeken en wisselen met het voorlaatste. Zo verder gaan tot de volledige tabel gerangschikt is. Selection sort rangschikt ter plaatse maar is niet stabiel. De definitieve gerangschikte tabel wordt element per element geconstrueerd: na iedere iteratie staan de laatste i elementen op de juiste plaats (in tegenstelling tot insertion sort, waar de defintieve plaats van ieder element maar gekend is op het einde).

2.3.3 Voorbeeld

Het grootste element is iedere keer rood ingekleurd en wordt verwisseld met het laatste nog niet grijze element.

2	3	1	5	4
2	3	1	4	5
2	3	1	4	5
2	1	3	4	5
1	2	3	4	5
1	2	3	4	5

2.3.4 Complexiteit

Selection sort is $\Theta(n^2)$.

2.3.5 Waarom Beter?

Deze methode wordt nauwelijks gebruikt, omdat ze eigenlijk gewoon slecht is. Als de selectieprocedure beter wordt kan het wel een goede methode opleveren.

2.3.6 Mogelijke Optimalisaties

Betere selectieprocedure. Heap Sort is hier een voorbeeld van.

2.3.7 Analyse/Bewijs complexiteit

Het aantal sleutelvergelijkingen is onafhankelijk van de volgorde van de elementen. Het aantal verwisselingen is hoogstens $n-1$ en dus $O(n)$. Het aantal keer dat het maximum aangepast wordt hangt samen met het aantal sleutelvergelijkingen

$$(n-1)+(n-2)+...+2+1=n(n-1)/2$$

Aantal keer dat het maximum aangpast wordt is dus $O(n^2)$. Dit zorgt er voor dat selection sort een $\Theta(n^2)$ algoritme is.

2.4 Heap Sort

2.4.1 Doel

Sorteren van elementen

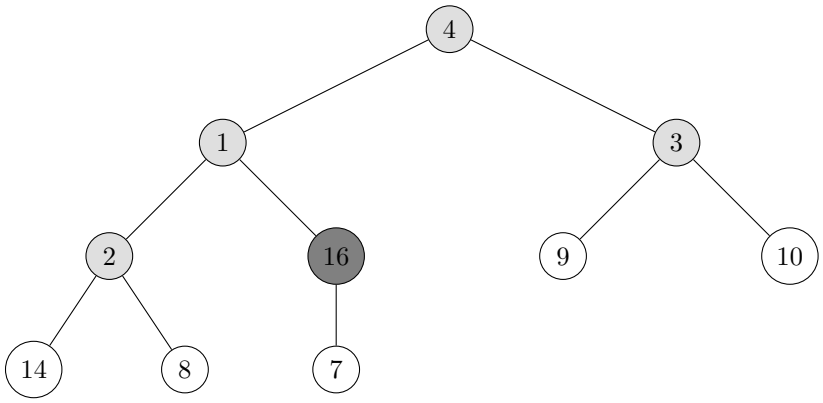
2.4.2 Basisprincipes

Eerst wordt de tabel getransorfmeerd tot *heap*. Nu kan het maximum van de tabel veel efficiënter gevonden worden (het is namelijk altijd het eerste element in de tabel). Door nu telkens de wortel van de heap te wisselent met het laatste element van de heap wordt de gerangschikte rij geconstrueerd. Na iedere wissel moet de *heapvoorwaarde* wel hersteld worden. Voor meer informatie over de operaties op *heaps* en hoe ze opgesteld worden verwijs ik naar het deel Heaps bij Gegevensstructuren I. Heap sort rangschikt ter plaatse maar is niet stabiel. De volgorde van gelijke elementen kan veranderen bij de heapconstructie en herstellen *heapvoorwaarde*.

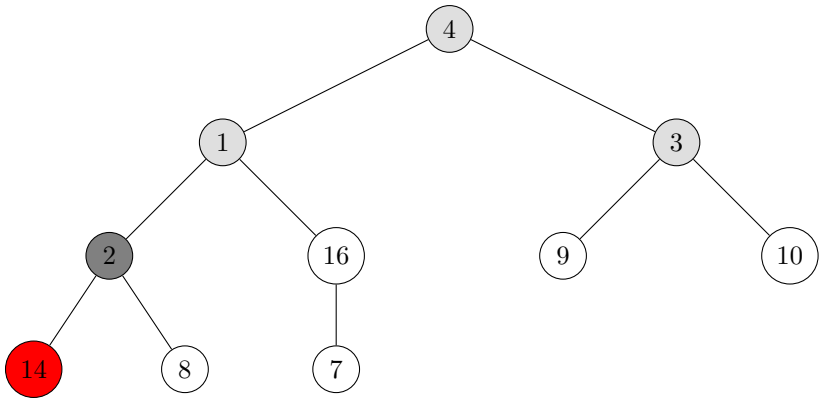
2.4.3 Voorbeeld

4	1	3	2	16	9	10	14	8	7

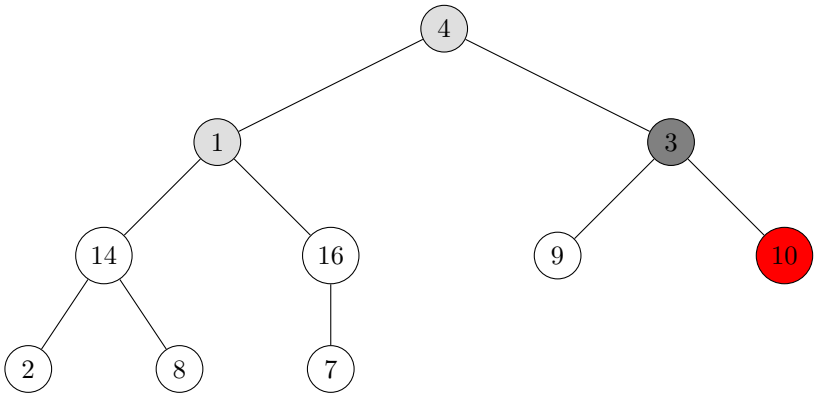
Deze tabel wordt eerst omgezet naar een heap door samenvoegen van deelheaps. We beginnen met volgende boom:



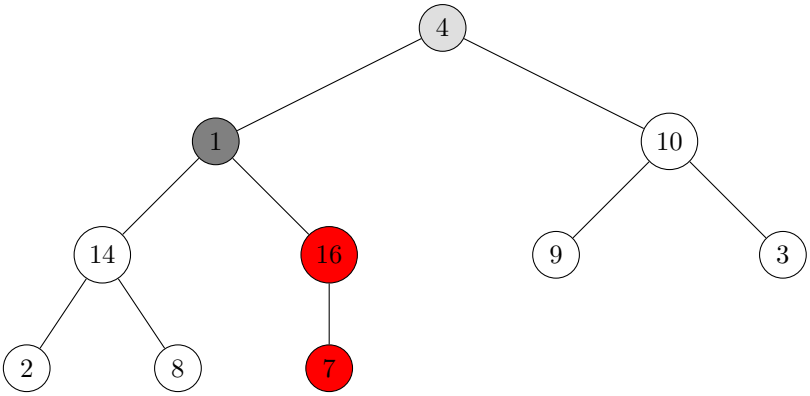
De nog grijs ingekleurde cellen moeten eventueel nog zakken om aan de heapvoorwaarde te voldoen. We beginnen met de knoop met waarde 16. Deze heeft geen kinderen met een grotere waarde. Volgende knoop is deze met waarde 2, deze heeft een kind met grotere waarde, we wisselen altijd met het kind met de grootste waarde.



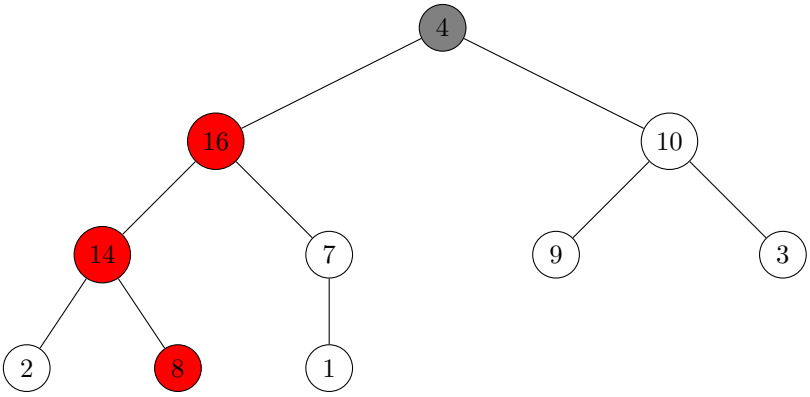
De volgende knoop is deze met waarde 3. Deze wordt gewisseld met deze met waarde 10.



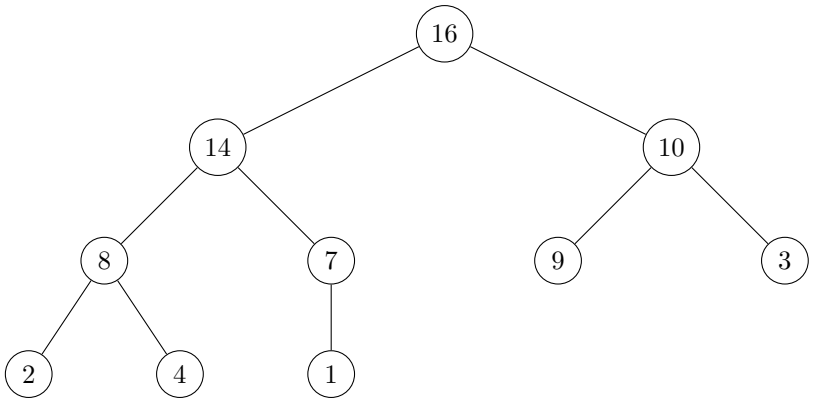
De volgende knoop met waarde 1 moet meerdere plaatsen zakken (zolang hij kinderen heeft moet grotere waarde wordt er gewisseld).



Als laatste laten we ook de wortel zakken tot zijn correcte plaats.



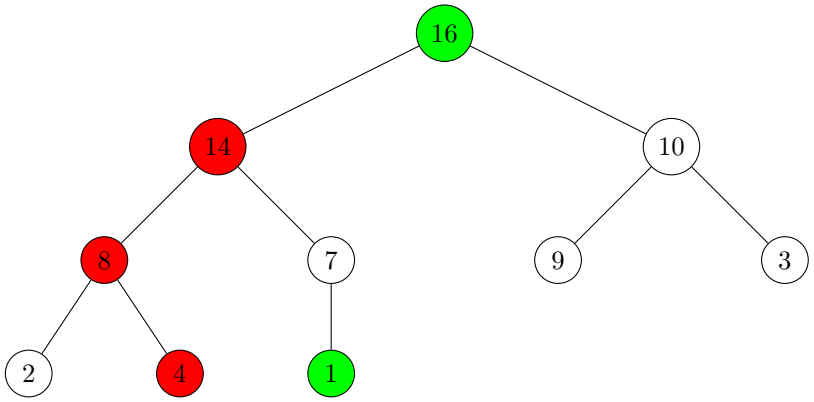
De heap wordt uiteindelijk:



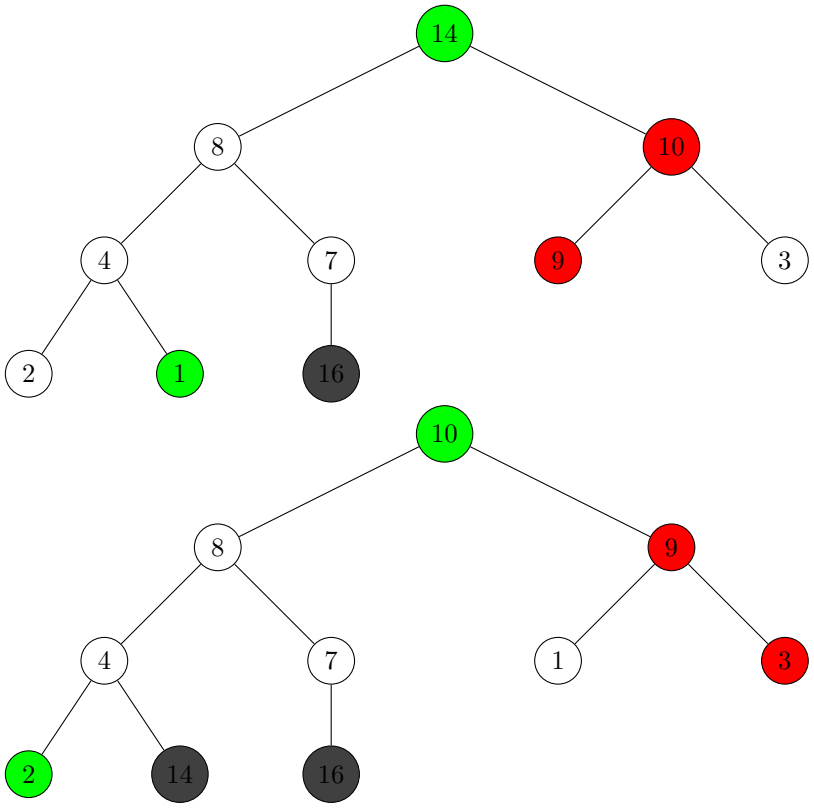
De tabel ziet er nu als volgt uit:

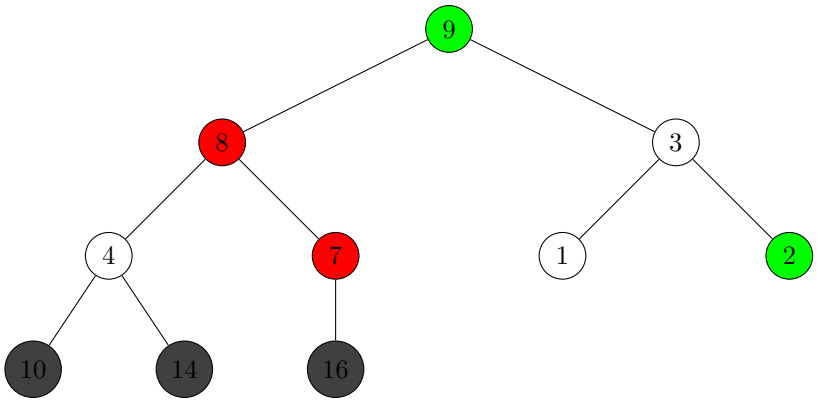
16	14	10	8	7	9	3	2	4	1

Nu kan de gerangschikte tabel efficiënt geconstrueerd worden door steeds de wortel van de heap met het laatste blad te wisselen, de heap te verkleinen en dan de *heapvoorwaarde* te herstellen. De knoop met waarde 1 zal zakken via de in het rood aangeduide weg.



Dit proces gaat verder tot de gerangschikte tabel volledig geconstrueerd is.





Ik ga hier geen 25 pagina's vol van die stomme bomen zetten. *Hope you get it by now.*

2.4.4 Complexiteit

Rangschikken verwijdert $n-1$ keer het wortelelement van de heap. De reconstructie van de heap is iedere keer $O(\lg n)$. Dit maakt heap sort $O(n \lg n)$. Gemiddelde geval is ook $\Theta(n \lg n)$ (zonder bewijs), testen tonen aan dat dit gedrag zeer consistent is: gemiddelde geval nauwelijks sneller dan slechtste.

2.4.5 Waarom Beter?

Voor grote n is heap sort sneller dan eenvoudige methodes.

2.5 Merge Sort

2.5.1 Doel

Sorteren van elementen

2.5.2 Basisprincipes

Merge sort gebruikt de 'verdeel en heers' techniek om een rangschik probleem op te lossen. Ze verdeelt het probleem in een aantal *onafhankelijke* deelproblemen. Merge sort zal de tabel iedere keer in twee delen en de deeltabellen sorteren. Dit blijft zo doorgaan tot een tabel met grootte één, deze is namelijk al gesorteerd. Om twee tabellen samen te voegen wordt iedere keer de kleinste elementen van de tabellen te vergelijken en het minimum weg te nemen. Als het einde van één van de deeltabellen bereikt wordt kan de rest van de andere zo overgenomen worden. Merge sort sorteert dus niet ter plaatse, er is een hulptabel nodig met als grootte de grootte van de kleinste deeltabel. Merge sort is wel stabiel zolang bij gelijke kleinste waarden altijd de linkse waarde genomen wordt als minimum.

Er bestaat ook een niet recursieve versie. Deze gebruikt een *bottom-up* verwerking. De tabel van begin tot einde overlopen, waarbij alle paren opeenvolgende elementen samengevoegd worden tot deeltabellen van lengte twee. Dan opnieuw en samenvoegen tot lengte vier, enz.

2.5.3 Voorbeeld

Verdelen van tabel in deeltabellen:

Samenvoegen deeltabellen:

2.5.4 Complexiteit

Best-, *Worst-* en *Average-case* zijn allemaal hetzelfde. Merge sort is dus $\Theta(n \lg n)$

2.5.5 Waarom Beter?

Mergesort is asymptotisch even efficiënt als heap sort, maar blijkt iets sneller te zijn met een goede implementatie. Oorspronkelijke volgorde van de elementen speelt nauwelijks een rol voor deze methode, wat haar gedrag dus heel consequent maakt: *Best-*, *Worst-* en *Average-case* zijn allemaal hetzelfde.

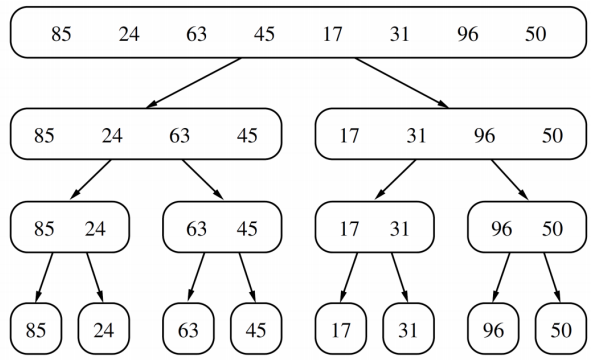


Figure 2: Verdelen tabel in deeltabellen

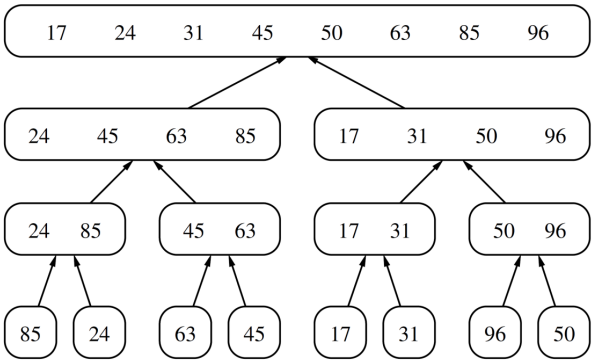


Figure 3: Samenvoegen deeltabellen

2.5.6 Mogelijke Optimalisaties

- Insertion Sort gebruiken als de deeltabellen klein genoeg zijn. Het is efficiënter dan de volledige recursie opnieuw te doen voor kleine tabellen. Insertion sort zorgt er ook niet voor de het algoritme niet meer stabiel is.
- Als er een hulptabel van grote n is, kan er afwisselend samengevoegd worden. Waardoor nutteloos kopiëren voor iedere *merge* niet meer nodig is.

2.5.7 Analyse/Bewijs complexiteit

- *Recursieve versie*:

2.6 Skelet

2.6.1 Doel

Sorteren van elementen

2.6.2 Basisprincipes

2.6.3 Voorbeeld

2.6.4 Complexiteit

2.6.5 Waarom Beter?

2.6.6 Mogelijke Optimalisaties

2.6.7 Analyse/Bewijs complexiteit

3 Gegevensstructuren I

3.1 Heaps

Een heap is een *complete binaire boom*, waarvan de elementen voldoen aan de *heapvoorwaarde*. De *heapvoorwaarde* van een *stijgende* heap (of *maxheap*) zegt dat de sleutel van de ouder minstens even groot moet zijn als deze van zijn kind(eren). De onderlinge volgorder van de sleutels van de kinderen maakt hierbij niet uit. De kinderen van iedere knoop i zijn te vinden op indexen $2i+1$ en $2i+2$. De ouder van een knoop is te vinden op index $\lfloor (i-1)/2 \rfloor$. De hoogte van een heap $h = \lfloor \lg n \rfloor$

3.1.1 Operaties

- *Element toevoegen.* Nieuwe knoop maken op laagste niveau (**bij tabelindex** $n+1$). Nu moet enkel de heapvoorwaarde nog herstelt worden: Zolang de nieuwe knoop een ouder heeft met een grotere sleutel wisselen deze knopen. Zodat de nieuwe knoop stijgt in de heap tot hij een ouder heeft met een grotere of even grote sleutel. Deze methode kan vergeleken worden met insertions sort: Kleinere elementen worden ook opgeschoven, maar deze keer wel via de weg in de heap tussen wortel en de nieuwe knoop.

- *Wortelelement vervangen* door nieuwe knoop g. De *heapvoorwaarde* kan verbroken worden als de nieuwe waarde kleiner is. In dit geval wisselen we de knoop met zijn grootste kind. Zo laten we de nieuwe waarde zakken in de heap tot op zijn correcte plaats.

- *Wortelelement verwijderen.* Deze operatie komt neer op het vervangen van de wortel door het laatste element in de kleiner geworden heap.

- *Element van een willekeurige knoop vervangen.* De *heapvoorwaarde* kan maar in één richting verstoord worden. Richting de wortel als de nieuwe waarde groter is, dan is het analoog aan een nieuwe waarde toevoegen aan een heap. Als het nieuwe element kleiner is dan is de situatie analoog aan het wortelelement vervangen, maar dan toegepast op de deelheap.

3.1.2 Constructie

- *Door toevoegen.* Elementen één voor één toevoegen aan een oorsponkelijk ledige heap. Tabel kan ter plaatse getransformeerd worden tot heap.

- *Door samenvoegen van deelheaps.* Beschouw tabel direct als *complete, binaire boom* en itereer over de eerste $\lfloor n/2 \rfloor$ elementen van rechts naar links. Als een van deze knopen een kind heeft met een grotere waarde blijven we deze knoop wisselen met zijn grootste kind tot de knoop geen kinderen meer heeft met een grotere waarde of een blad geworden is.

Voorbeeld van constructie met samenvoegen.

3.1.3 Sterktes en Zwaktes

- Meeste bewerkingen op een heap zijn efficiënt omdat hun tijdsduur begrenst wordt door de hoogte van de heap, die veel kleiner is dan het aantal elementen.

3.1.4 Complexiteit van Operaties

- *Element toevoegen.* Het aantal keer dat we knopen moeten wisselen is beperkt door de hoogte van de heap. Een element toevoegen heeft dus $O(h) = O(\lg n)$ operaties.

- *Wortelelement vervangen.* De langste weg langs waar de nieuwe wortel zou moeten zakken is nog steeds h dus de complexiteit blijft $O(h) = O(\lg n)$. Er moeten wel meer vergelijkingen gebeuren (knoop heeft max 2 kinderen en max 1 ouder).

- *Wortelelement verwijderen.* Aangezien dit analoog is aan het wortelelement vervangen is deze operatie ook $O(\lg n)$.

- *Element van willekeurige knoop vervangen.* Ook hier is de grootste afgelegde weg van een vervangen knoop h. De complexiteit van deze operatie is dus ook $O(h) = O(\lg n)$

- *Constructie door toevoegen.* Er moet $n-1$ keer een element toegevoegd worden aan de heap (het eerste element staat al goed). In het slechtste geval moet iedere knoop die toegevoegd wordt de volledige hoogte van de heap doorlopen (nieuwe wortel worden). Als dan ook nog eens het laatste niveau volledig opgevuld wordt krijgen we een bovengrens voor performantie. Het aantal keer dat gewisseld wordt per niveau kan uitgedrukt worden ifv de hoogte: $2^h h$. Het totaal aantal wissels voor de volledige heap is dus maximaal $\sum_{i=0}^{h} 2^h h$. Door volgende afschatting te gebruiken kan de complexiteit voor het opstellen van de heap berekend worden:

$$\sum_{i=0}^{h} 2^h h \leq 2^{h+1} h = 2^{\lg n+1} \lg n = \lg n(n+1) = O(n \lg n)$$

- *Constructie door samenvoegen.* Veel efficiënter dan toevoegen. Heap wordt van onder naar boven opgebouwd. In het slechtste geval moeten alle eerste $\lfloor n/2 \rfloor$ elementen naar bladeren van de boom gebracht worden. Het aantal wissels per niveau is $2^i(h-i)$. Het maximale aantal wissels is dus $\sum_{i=0}^{h-1} 2^i(h-i)$. Ook hier kan een afschatting gebruikt worden om de complexiteit te berekenen:

$$\sum_{i=0}^{h-1} 2^i(h-i) \leq 2 \cdot 2^{h+1} = 2 \cdot 2^{\lg n+1} = 2(n+1) = O(n)$$

3.1.5 Toepassingen

Heap Sort

3.2 GegSkelet

3.2.1 Operaties

3.2.2 Sterktes en Zwaktes

3.2.3 Complexiteit van Operaties

3.2.4 Analyse/Bewijs Complexiteit van Operaties

3.2.5 Toepassingen

4 Graafalgoritmen I