Datastructuren en Algoritmen II, 2016

Gunnar Brinkmann

19 september 2016

Inhoudsopgave

1 Introductie

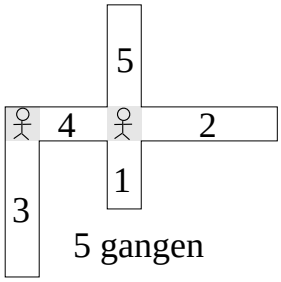
Volgens mij bevat de les *Datastructuren en Algoritmen II* heel leuke en interessante ideeën! (Dat mag ik zeggen, omdat de meeste ideeën natuurlijk niet van mij zijn, maar van andere mensen en ik ze alleen aan jullie voorstel.) Maar toch vinden sommige studenten de les niet zo leuk als ik zou hopen en/of verwachten.
De reden is dat de les soms als *te wiskundig* wordt beschouwd. Daardoor hebben studenten die grote problemen hebben met het deel dat zij als wiskunde zien, soms moeite om de mooie ideeën te waarderen – die staan dan een beetje in de achtergrond.
Maar inderdaad worden in deze les geen bijzonder ingewikkelde wiskundige theorieën of methoden gebruikt. Wat wel gebruikt wordt, is *zorgvuldig redeneren* – en als iemand zorgvuldig redeneert en daardoor iets bewijst dan wordt dat soms als wiskunde gezien. Maar *bewijzen* – of met andere woorden *absoluut zeker zijn dat iets klopt* – is niet alleen in de wiskunde nuttig! De bedoeling is gewoon *fouten te vermijden*, en dat fouten in algoritmen (meestal – zie hoofdstuk 9) vermeden moeten worden, denken jullie hopelijk ook.

Maar misschien is het het beste als wij naar een voorbeeld kijken. De volgende vraag stond ongeveer zo in een examen:

Opgave:

In een museum zijn er verschillende gangen die bewaakt moeten worden. De

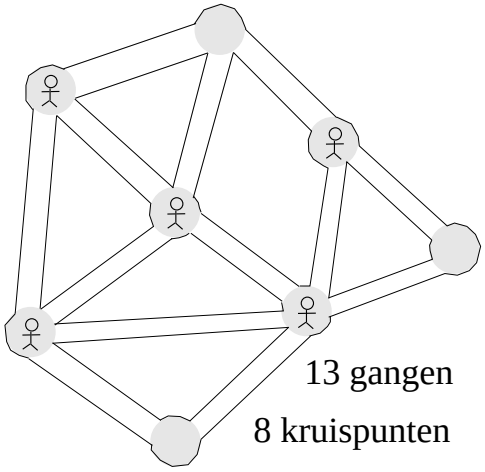
gangen zijn allemaal recht en relatief kort zodat iemand die op het einde van een gang staat de hele gang kan zien en snel kan bereiken. Het eindpunt van een gang kan tegelijk het eindpunt van andere gangen zijn – dat noemen wij dan een kruispunt. Wij stellen dat er ten minste 2 gangen zijn en dat elke gang aan ten minste één kruispunt grenst.



5 gangen

2 kruispunten

4 eindpunten die
geen kruispunten zijn



13 gangen

8 kruispunten

De taak is nu een manier te vinden om bewakers op de eindpunten van de gangen te plaatsen zodat het aantal bewakers zo klein mogelijk is, maar elke gang door ten minste één bewaker gezien en snel bereikt kan worden.

Veel studenten zijn hun algoritme als volgt begonnen:

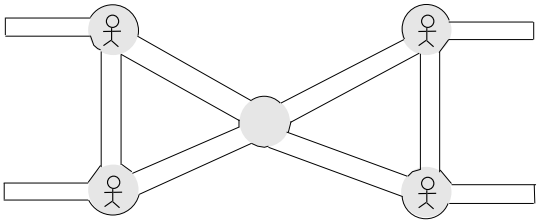
- Plaats een bewaker op een kruispunt waar de meeste gangen samenkomen.

Dit zou een begin zijn van wat wij later een gretig algoritme zullen noemen (hoofdstuk 8.1).
Maar je moet natuurlijk argumenteren waarom dat een goede keuze is. Het meest gebruikte (maar niet echt formele) argument is: *dat is toch duidelijk!*

Iets formeler zou je kunnen argumenteren: als de eerste bewaker op een ander kruispunt wordt geplaatst dan zijn er achteraf even veel of zelfs nog meer onbewaakte gangen dan met deze keuze. En omdat meer onbewaakte gangen ook even veel of meer bewakers vragen, kunnen oplossingen zonder een bewaker op dit punt nooit beter zijn.

Dat is ook niet echt een bewijs, maar misschien toch iets overtuigender – of niet?

Maar jammer genoeg klopt het niet. Kijk eens naar de volgende tekening:



Het is gemakkelijk om te zien dat er een uniek kruispunt is waar het grootste aantal gangen samenkomt. Maar geen enkele van de optimale oplossingen (er is er in feite maar één) plaatst een bewaker op dit kruispunt. Er is dus duidelijk iets fout met de informele argumentatie!
Het zou best kunnen dat jullie later jullie eigen algoritmen moeten ontwerpen en dat een fout algoritme best een grote schade voor jouw werkgever kan veroorzaken. In gevallen waar het iets met veiligheid te maken heeft, kan het zelfs nog veel erger zijn! Het zorgvuldig – of jullie kunnen het ook wiskundig noemen – redeneren gaat dus niet over wiskunde maar over *zorgvuldig werken* en proberen garanderen dat dingen waarvan je denkt dat die kloppen **werkelijk** juist zijn...

Een voorbeeld van een iets formelere redenering is bv. het bewijs van de volgende opmerking:

Opmerking:

Als er een gang g is die aan een enkel kruispunt ligt, dan is er altijd een optimale oplossing met een bewaker op dit kruispunt. Het is dus nooit nodig een bewaker op het einde van een gang te plaatsen als er geen kruispunt is.
Bewijs:
Stel dat er een optimale oplossing bestaat die een bewaker b op het einde van g plaatst dat geen kruispunt is. Wij weten dat het andere einde van g wel een kruispunt is en dat daar geen bewaker staat: anders zou g door twee bewakers bewaakt worden en omdat b **alleen maar** g bewaakt zouden wij hem gewoon kunnen verwijderen en een betere oplossing hebben. Maar dat kan natuurlijk niet, omdat de oplossing al optimaal was (een tegenstrijdigheid). Maar dan kunnen wij b verplaatsen naar het kruispunt en hebben we en oplossing die aan de voorwaarden voldoet en even veel bewakers gebruikt en dus ook optimaal is!

Na dit bewijs zijn jullie dus **absoluut zeker** dat het juist is als een algoritme nooit probeert een bewaker op een punt te plaatsen dat geen kruispunt is!
Soms is het moeilijk te zien, wat een *juist bewijs* is en wat een informele en misschien zelfs foute redenering is – dat moeten jullie gewoon oefenen. Maar ik hoop dat jullie akkoord gaan dat het daarbij niet om wiskunde gaat maar om informatica.

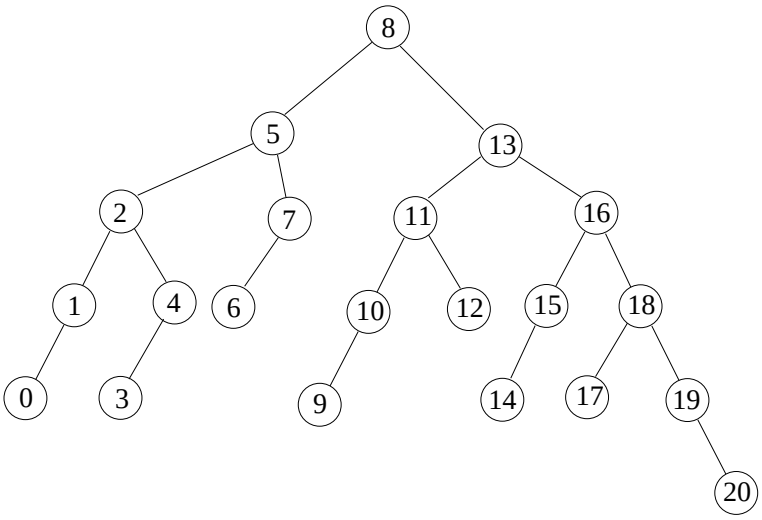
De bedoeling van deze tekst is **niet** in plaats van de les te worden gebruikt maar alleen om te helpen de eigen nota's misschien een beetje beter te verstaan als er iets niet echt duidelijk is. Alleen in de les kan je onmiddellijk vragen stellen als je iets niet verstaat...
Belangrijk: De basis van het examen zijn de les en de oefeningen. Als er een resultaat, algoritme, datastructuur, techniek, etc. in de les gegeven wordt en niet in de lesnota's staat, kan het nog altijd deel uitmaken van het examen. Het is dus belangrijk naar de les te gaan en in gevallen waar dat niet mogelijk is ervoor te zorgen dat je weet wat er gegeven werd. Deze keer gaat er vermoedelijk ook een oefening in het examen voorkomen die heel sterk op een in de oefeningenles besproken oefening lijkt. Dit is nog een reden meer om naar de oefeningenles te gaan en vragen te stellen om de oplossingen van de oefeningen goed te verstaan!
Daarbij is het in geen enkel geval zinvol bewijzen of stellingen gewoon *uit het hoofd te leren*. De bedoeling is de bewijzen, technieken, stellingen, algoritmen, datastructuren en ideeën **te verstaan** en dan ook in een andere context te kunnen toepassen. Jullie kunnen de examens van de laatste jaren online vinden en jullie zullen zien dat in geen enkel ervan bewijzen werden gevraagd die **precies zo** ook deel uitmaakten van de les – maar wel bewijzen waar je technieken moest toepassen die ook in bewijzen in de les gebruikt werden.

In de volgende lesnota's zullen wij het vaak over *datastructuren* hebben. Daarbij is het niet mogelijk de *structuren* en de bewerkingen op deze structuren van elkaar te scheiden. Een datastructuur is dus een *structuur* (bv. een boom) **samen** met zekere bewerkingen op deze boom. Een *semi-splay boom* is bv. een heel gewone zoekboom – maar het wordt verondersteld dat de bewerkingen op de boom altijd de *semi-splay* bewerkingen zijn. Alle uitspraken die wij over de complexiteit van datastructuren maken, gelden dus voor de beschreven bewerkingen die wij als deel van de datastructuur beschouwen.

2 Zoekbomen

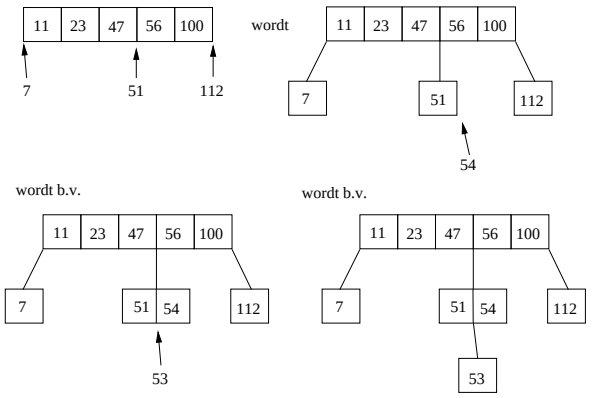
In DA I hebben jullie al de definitie van een binaire zoekboom gezien en hebben jullie ook een voorbeeld gezien hoe je ervoor kan zorgen dat een zoekboom niet te slecht gebalanceerd is – namelijk AVL-bomen van G.M. Adelson-Velskii en E.M. Landis. Toen hebben jullie rotaties gebruikt om de zoekboom te herbalanceren. In deze les zien wij verschillende andere technieken om ervoor te zorgen dat de bomen goed gebalanceerd zijn en niet alle bomen zullen binaire bomen zijn.

Oefening 1 *Een herhaaloefening:*
Verwijder sleutel 6 uit deze AVL-boom:



Waarom gebruiken wij zoekbomen en niet bv. gesorteerde lijsten? Als je een gesorteerde arraylist hebt, kan je heel mooi en efficiënt sleutels opzoeken (logaritmisch zoeken). Het probleem is gewoon dat je als je een sleutel wil toevoegen die niet op het einde moet zitten, soms dingen moet verschuiven. Dat wordt opgelost door *linked lists* (of *geschakelde lijsten*). Probleem: in linked lists kan je wel gemakkelijk iets tussenvoegen, maar het zoeken wordt ingewikkelder. Wat kunnen wij dus doen als wij aan de ene kant geen gelinkte lijst willen gebruiken om efficiënt te kunnen zoeken en aan de andere kant als wij iets vóór de eerste sleutel of tussen 2 sleutels willen invoegen en daarbij deze sleutels niet willen verplaatsen? Een oplossing is gewoon tussen de toppen een referentie aan te brengen die zegt waar de toppen die ertussen moeten staan te vinden zijn. Dat zie je bv. in figuur 1. Als je dat meerdere keren toepast dan zie je dat er op een heel natuurlijke manier bomen ontstaan. Je kan zoekbomen dus begrijpen als een manier om het toevoegen

aan gesorteerde lijsten efficiënter te maken. Binaire zoekbomen zijn maar een speciaal geval ervan: het geval waar in elke lijst maar 1 sleutel mag staan.



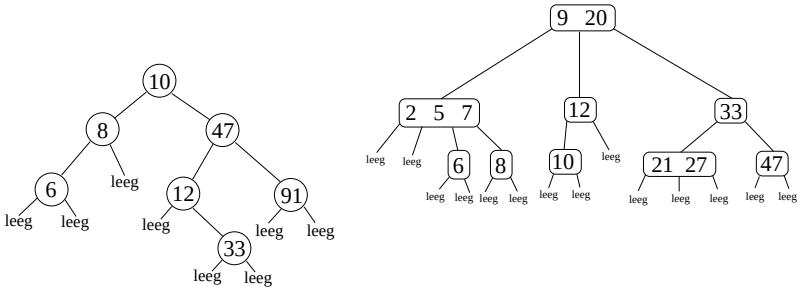
Figuur 1: In een gesorteerde lijst iets *tussen bestaande sleutels duwen*.

2.1 Deelbomen vervangen

Omdat niet alleen maar binaire bomen als zoekbomen geschikt zijn, geven wij eerst een iets algemenere definitie van een zoekboom. Soms beschrijven wij NULL-pointers als *lege toppen* of *lege kinderen* om niet altijd een verschil te moeten maken tussen een bestaand kind en een niet-bestaand kind. Als jullie het zouden implementeren, zouden jullie natuurlijk geen top aanmaken als er geen sleutel inzit, maar gewoon een NULL-verwijzing gebruiken.

Definitie 1 *Een zoekboom is een boom met een gegeven wortel w. Een top t bevat sleutels $s_1, \ldots, s_{n(t)}$ waarbij het aantal $n(t) \geq 1$ van de top t afhangt en $s_1 < s_2 < \cdots < s_{n(t)}$.*
Als top t precies $n(t)$ sleutels bevat dan heeft t precies $n(t)+1$ verwijzingen naar kinderen $k_1, \ldots, k_{n(t)+1}$ waarbij de indices een volgorde aangeven. Sommige van de verwijzingen mogen leeg zijn (dus NULL-pointers).
Als wij definiëren $s_0 = -\infty$ en $s_{n(t)+1} = \infty$ dan geldt voor $1 \leq i \leq n(t)+1$ en elke sleutel s in de deelboom T_i met wortel k_i: $s_{i-1} < s < s_i$.
Voor een gegeven i noemen wij de T_j met $1 \leq j \leq i$ soms de kleinere kinderbomen *van s_i en de T_j met $i < j \leq n(t)+1$ soms de* grotere kinderbomen*.*

Een binaire zoekboom is dus gewoon een zoekboom met maar 1 sleutel per top dus $n(t) = 1$ voor elke top.
Omdat wij het in dit deel toch al alleen maar over zoekbomen zullen hebben, schrijven wij soms alleen *boom* als wij een zoekboom bedoelen.



Figuur 2: Twee zoekbomen.

Definitie 2 *Stel dat T een zoekboom is en T' een deelboom van T.*
Als x een top in T' is en een verwijzing naar een top $y \notin T'$ bevat, dan noemen wij de maximale deelboom van T met wortel y een buitenboom van T'. Als x een lege verwijzing bevat dan noemen wij dat ook een (lege) buitenboom. Een deelboom kan dus meerdere (verschillende) lege buitenbomen hebben.
Voor de verzameling van alle buitenbomen van T' schrijven wij $B(T')$.

Daarbij betekent *de* **maximale** *deelboom met wortel y* dat de deelboom samen met de wortel y ook **alle** opvolgers van y bevat (dus de kinderen, de kinderen van de kinderen, etc. tot de bladeren).

Definitie 3 *Gegeven een zoekboom T, en een deelboom T'.*
Voor buitenbomen $b, b' \in B(T'), b \neq b'$ definieer $b < b'$ als en slechts als er een sleutel x in T' zit waarvoor b deel uitmaakt van een kleinere kinderboom en b' deel uitmaakt van een grotere kinderboom.
Zie Oefening 2 voor het bewijs dat dat in feite een goede definitie is.

Oefening 2

Gegeven een zoekboom T en een deelboom T' van T.

a.) *Toon aan dat als T' een deelboom met s sleutels is dat dan het aantal buitenbomen van T' altijd $s + 1$ is – om het even wat de structuur van T' is.*

b.) *Toon aan dat er voor gegeven buitenbomen $b, b' \in B(T'), b \neq b'$ precies één top bestaat die één of meerdere sleutels met de eigenschap uit Definitie 3 bevat.*

Toon aan dat dit een totale orde "$<$" op de verschillende elementen van $B(T')$ definieert.

c.) *Geef een alternatieve definitie voor de orde "$<$" in b.) die de sleutels in de ouders van de buitenbomen gebruikt of sleutels die in de buitenboom zouden terechtkomen (waarbij je mag stellen dat de elementen van de boom uit $\mathbb{Q}$ zijn, en je dus altijd een element kan vinden dat in de boom terechtkomt). Toon aan dat de definities inderdaad equivalent zijn.*

De rotaties die jullie in DA I hebben gezien, zijn inderdaad maar een speciaal geval van een veel algemenere techniek:

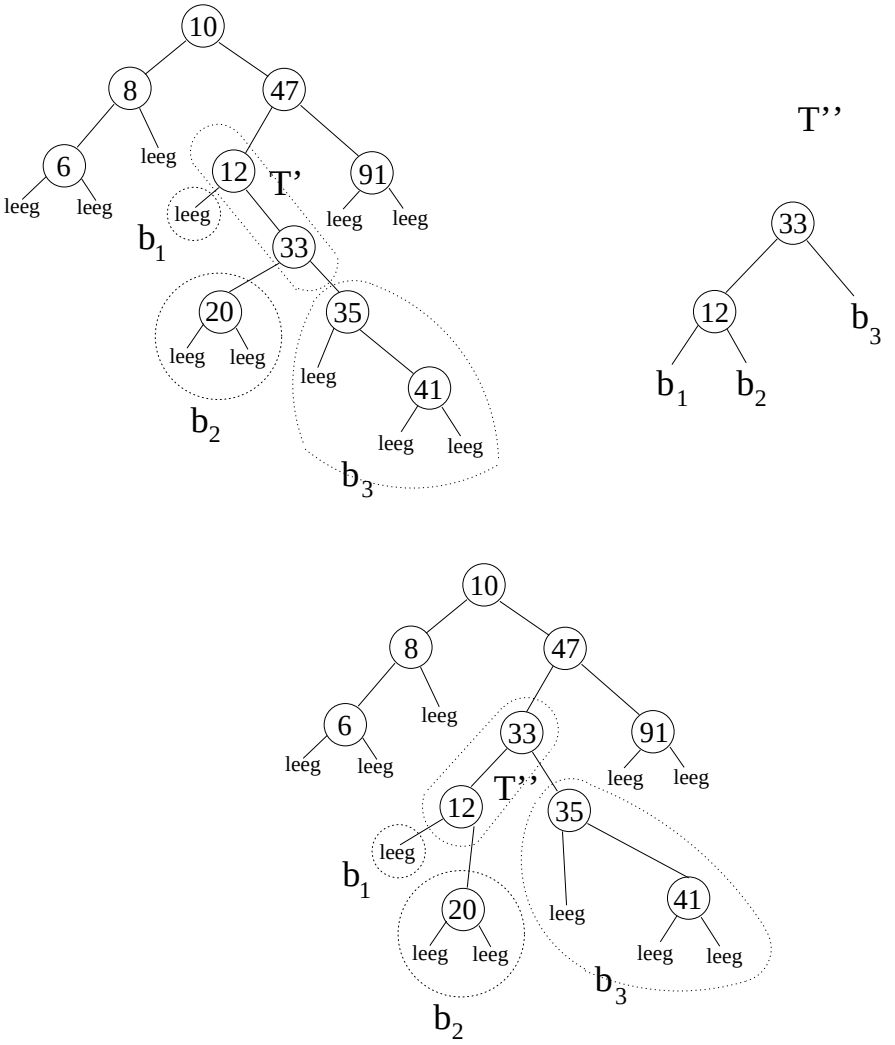
Algoritme 1 *Gegeven een zoekboom T en een deelboom T' met wortel w'. Stel dat de buitenbomen van T' de bomen $b'_1 < \cdots < b'_k$ zijn.*
Bovendien zij T'' met wortel w'' een zoekboom die precies dezelfde sleutels bevat als T'. T'' heeft dus k lege buitenbomen $b''_1 < \cdots < b''_k$.
Nu kan je het volgende doen:

- *verwijder T' uit T*
- *plaats T'' als nieuwe deelboom van T:*
  - *als w' de wortel van de hele boom T was, maak w'' de nieuwe wortel – anders laat de pointer die vroeger naar w' wees nu naar w'' wijzen.*
  - *als een verwijzing in T'' naar b''_i wees, laat hem dan naar b'_i wijzen (dus: plak de buitenbomen van T' in volgorde aan T'').*

Dan is de nieuwe boom $\bar{T}$ een zoekboom met dezelfde sleutels als T. Terwijl het duidelijk is dat $\bar{T}$ dezelfde sleutels bevat, moet wel bewezen worden dat het opnieuw een zoekboom is – maar eerst een voorbeeld.

In Figuur 3 zie je hoe een heel kleine deelboom vervangen wordt. Inderdaad bevat die maar twee toppen en is het resultaat precies hetzelfde als wat jullie al gezien hebben – het is een rotatie. Maar je kan ook grotere deelbomen vervangen. Dat zie je in Figuur 4 waar de boom T'' en dus ook het resultaat na het vervangen niet binair is.
Om aan te tonen dat het algoritme juist werkt, gebruiken wij 2 lemma's:

Lemma 1 *Gegeven een zoekboom T en een deelboom T' met sleutels $s_1 < \cdots < s_k$ en buitenbomen $b_1, \ldots, b_{k+1}$.*
Als wij definiëren $s_0 = -\infty$ en $s_{k+1} = \infty$ dan geldt voor elke $1 \leq i \leq k+1$ en elke sleutel $s \in b_i$: $s_{i-1} < s < s_i$.

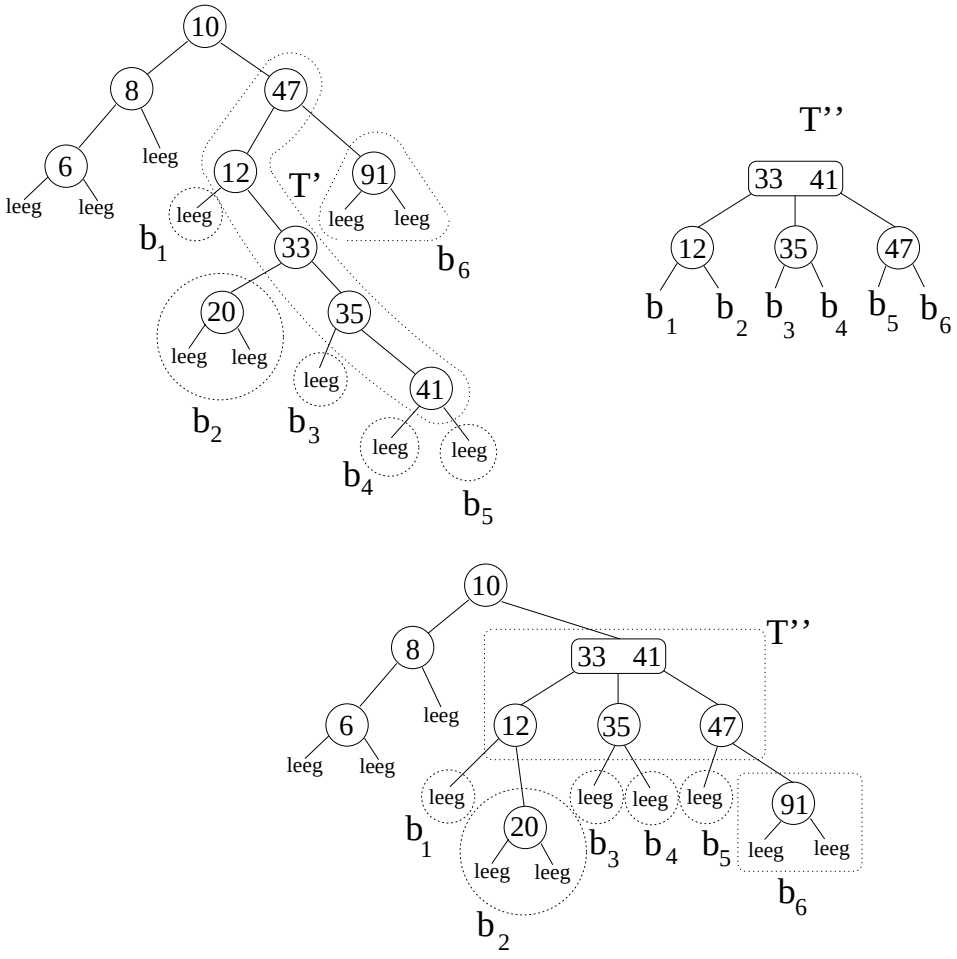


Figuur 3: Het vervangen van een heel kleine deelboom door een andere. Inderdaad is het resultaat hetzelfde als van een rotatie

Dat lijkt sterk op de eisen voor een zoekboom en inderdaad volgt het onmiddellijk als de boom maar 1 top bevat. In het algemeen kan je het misschien het best begrijpen door hele deelbomen gewoon te interpreteren als een manier om een enkele top voor te stellen.

Bewijs: Wij gebruiken inductie in het aantal n van toppen van de deelboom. Voor $n = 1$ – dus als alle sleutels in dezelfde top zitten – is het precies de definitie van een zoekboom. Stel nu dat $n \geq 2$.

Stel dat b een blad van T' is met sleutels $s_1^b, \ldots, s_k^b$ en kinderen $k_1^b, \ldots, k_{k+1}^b$. Zijn ouder noemen wij o en stel dat de sleutels in de ouder $s_1^o, \ldots, s_m^o$



Figuur 4: Het vervangen van een iets grotere deelboom. Het resultaat is geen binaire zoekboom.

zijn en de kinderen $k_1^o, \ldots, k_{m+1}^o$. Stel bovendien dat b kind k_i^o is. Dit alles kan je *stellen* omdat *stellen* hier alleen maar betekent dat een naam wordt gekozen. Dus geldt voor alle sleutels s in de deelboom met wortel b dat $s_1^o, \ldots, s_{i-1}^o < s$ en $s_i^o, \ldots, s_m^o > s$.

Als wij dus een nieuwe top o' in plaats van o aanmaken met sleutels $s_1^o, \ldots, s_{i-1}^o, s_1^b, \ldots, s_k^b, s_i^o, \ldots, s_m^o$ en kinderen $k_1^o, \ldots, k_{i-1}^o, k_1^b, \ldots, k_{k+1}^b$, $k_{i+1}^o \ldots k_{m+1}^o$ dan is het resultaat een nieuwe zoekboom (kan je dat aantonen?) en heeft de zo geconstrueerde deelboom T'' dezelfde sleutels en dezelfde buitenbomen in dezelfde volgorde. (Kan je formeel aantonen dat de volgorde van de buitenbomen inderdaad dezelfde is?)

Maar de nieuwe zoekboom heeft één top minder, met inductie volgt

dus de stelling.

■

Maar inderdaad geldt ook de andere richting:

Lemma 2 *Gegeven een zoekboom T' met sleutels $s_1 < \cdots < s_k$ en zoekbomen $b_1, \ldots, b_{k+1}$ met de eigenschap dat voor elke $1 \leq i \leq k+1$ en elke sleutel $s \in b_i$ geldt $s_{i-1} < s < s_i$ (waarbij opnieuw $s_0 = -\infty$ en $s_{k+1} = \infty$).*
Als wij dan voor $1 \leq i \leq k+1$ de boom b_i de i-de buitenboom van T' maken dan is het resultaat een zoekboom.

Bewijs: Het bewijs lijkt sterk op het vorige bewijs, dus werken wij het niet in detail uit. Voor één top is het opnieuw gewoon de definitie. Dan kan je inductie gebruiken door twee toppen in T' samen te voegen en zo een kleinere boom T'' te bouwen. De boom die je krijgt als je de buitenbomen aan T'' plakt, is een zoekboom (volgens inductie). Door hem te vergelijken met wat je krijgt als je de buitenbomen aan T' plakt, kan je aantonen dat die ook aan de eigenschappen van een zoekboom voldoet.

■

In het vorige lemma is T' een hele zoekboom en niet een deel van een grotere zoekboom. Maar omdat als T' een deelboom van een grotere boom T is en ons deelboomvervangalgoritme de sleutels in de deelbomen niet verandert, voldoen die natuurlijk niet alleen aan de vereisten van Lemma 2 met $s_0 = -\infty$ en $s_{k+1} = \infty$ maar ook aan de grenzen die door de andere toppen op het pad van de wortel van T' naar de wortel van de hele boom gegeven worden. Voor deze toppen geldt dus nog altijd de zoekboomeigenschap. Dus:

Opmerking 3 *Als algoritme 1 op een zoekboom wordt toegepast, is het resultaat opnieuw een zoekboom.*

Als wij een boom willen herbalanceren moeten wij dus gewoon beschrijven welke deelboom wij verwijderen en welke deelboom wij in plaats van deze gebruiken.

Oefening 3 *Als wij een boom door een andere willen vervangen is het natuurlijk belangrijk dat op een efficiënte manier te doen. Als de deelbomen heel klein zijn, is dat gemakkelijk. Maar inderdaad is dat ook efficiënt mogelijk als de deelbomen groter zijn:*
Gegeven een arbitraire binaire zoekboom T met n toppen en een (andere of gelijke) binaire zoekboom T' die ook n sleutels bevat.

Beschrijf een algoritme dat in lineaire tijd (dus tijd $O(n)$) de boom T zo verandert (door veranderen van de links naar de kinderen) dat het resultaat T'' isomorf is met T' (maar nog altijd een zoekboom met dezelfde sleutels als vroeger is).

- *Definieer eerst wat* isomorf *precies betekent. Denk aan de cursus* Discrete Wiskunde*.*
- *Beschrijf het algoritme. Tip: bepaal eerst paren (x, y) met $x \in T, y \in T'$ van toppen zodat x in T'' de positie van y in T' moet krijgen. Maar je moet deze tip niet gebruiken – er zijn ook andere mogelijkheden om zo'n algoritme te ontwerpen...*
- *Toon aan, dat het algoritme juist is – dus dat de boom T'' die gebouwd wordt echt isomorf met T' is en dat het inderdaad een zoekboom is.*

Oefening 4 *In DA I hebben jullie de hersteloperaties voor AVL bomen geformuleerd door middel van enkelvoudige rotaties en dubbele rotaties. Hier hebben jullie gezien dat je deelbomen kan vervangen door andere deelbomen met dezelfde sleutels.*
Herschrijf de regels voor het herstellen van AVL-bomen door de deelboomvervangmethode te gebruiken. Geef gewoon elke keer de deelboom T die vervangen moet worden en de deelboom T' die hem vervangt.

Oefening 5 *Stel dat voor een zekere manier om zoekbomen te herbalanceren aangetoond kan worden dat een boom met diepte h ten minste $4^{h+2}/8$ toppen bevat.*
Toon aan dat de boom logaritmische diepte heeft (dus de diepte h is $O(\log t)$ met t het aantal toppen) en bepaal de constanten.

2.2 Semi-splay

Semi-splay is een techniek voor zelf-organiserende zoekbomen die in 1985 door Sleator en Tarjan uitgevonden werd. In hun artikel (dat zeker de moeite is om te lezen) introduceren ze verschillende zoekbomen, waaronder ook splay bomen en semi-splay bomen. Zij stellen voor splay bomen te gebruiken die iets ingewikkelder zijn dan semi-splay bomen omdat die volgens hun mening beter zouden presteren. Splay bomen waren dan ook de inhoud van veel lessen over algoritmen en werden in veel programma's gebruikt. In een artikel van 2009 werden de verschillende technieken dan op grote hoeveelheden data vergeleken en het bleek dat semi-splay niet alleen eenvoudiger was, maar ook bijna altijd efficiënter. Je moet al met opzet een reeks bewerkingen

gebruiken die op maat gemaakt zijn voor splay om ervoor te zorgen dat het beter presteert dan semi-splay.
In het geval van AVL-bomen (en andere zoekbomen die wij nog zullen zien) kan je garanderen dat elke bewerking van een boom met n sleutels in tijd $O(\log n)$ kan gebeuren. Hoewel we bij semi-splay niet altijd kunnen garanderen dat een bewerking in tijd $O(\log n)$ kan gebeuren, kunnen we wel garanderen dat een reeks van bewerkingen beginnend met een lege boom **gemiddeld** $O(\log n)$ tijd vraagt.
Als wij alleen naar deze eigenschap kijken, lijkt het veel verstandiger AVL-bomen te gebruiken omdat die ten slotte in elke stap goed presteren **en** dus ook gemiddeld. Maar splay-technieken hebben één voordeel: in de meeste toepassingen worden niet alle elementen in een zoekboom even vaak opgezocht en het zou natuurlijk mooi zijn als de elementen die vaak opgezocht worden dichter bij de wortel staan dan elementen die zelden worden opgezocht. Natuurlijk weet je niet op voorhand welke elementen vaak worden opgezocht en bovendien kan dat in de loop der tijd ook veranderen. Als je bv. een zoekboom met internetadressen hebt dan zoeken in december zeker weinig mensen naar de homepagina van de Tour de France maar in juni is dat zeker een van de meest opgezochte adressen. Een boom die er (op een of andere manier) rekening mee houdt hoe vaak een sleutel opgezocht wordt, kan in zulke gevallen dus beter presteren. Semi-splay doet dat door de boom zo te herbalanceren dat elke keer dat een sleutel wordt opgezocht hij iets dichter naar de wortel wordt verplaatst. Hoeveel dichter bij de wortel hij wordt geplaatst, hangt af van hoe het pad naar de sleutel er uitziet.
Het herbalanceren gebeurt door kleine deelboompjes te vervangen. Wij beschrijven de bottom-up manier, maar je kan het ook top-down implementeren.

Algoritme 2 **Semi-splay langs een pad.**

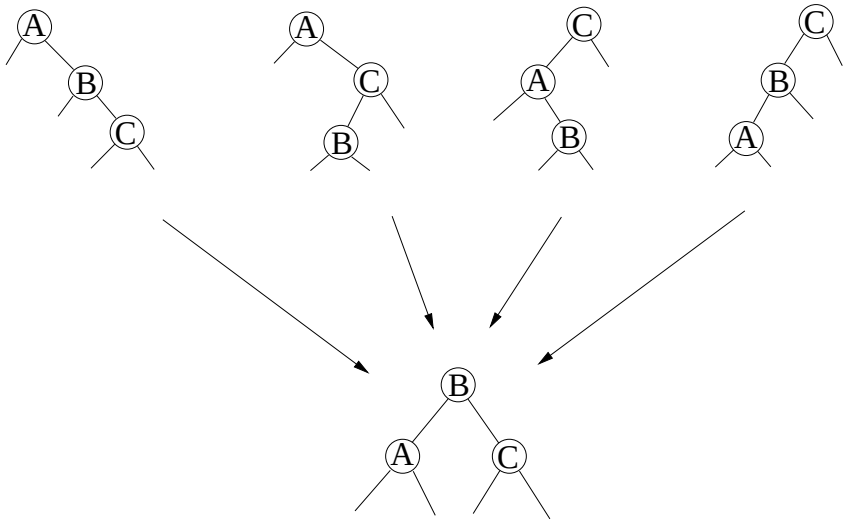
Stel dat een pad P vanaf de wortel naar een top t gegeven is.

Begin met t en herhaal dan de volgende stap tot t de wortel is of een kind van de wortel:

- *Ga op hetzelfde pad twee toppen terug. Nu bevat jouw deelpad 3 toppen. Vervang deze deelboom door een complete binaire boom met 3 toppen (zie afbeelding 5).*

  Neem de wortel van deze nieuwe deelboom als de nieuwe top t.

Semi-splay wordt elke keer toegepast als je een bewerking op een semi-splay-boom doet. Je moet alleen maar weten wat het pad is waarop je het moet



Figuur 5: Een pad met 3 toppen wordt vervangen door een complete binaire boom met 3 toppen.

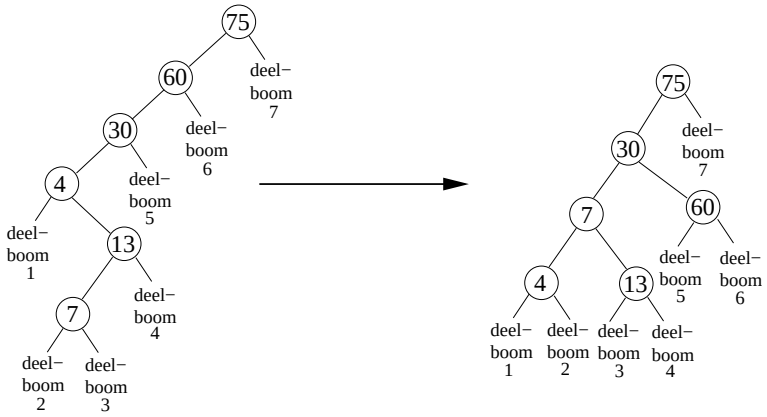
toepassen. Inderdaad zijn dat gewoon de paden die je tijdens een gewone bewerking in een binaire zoekboom (zoals gezien in DA I) doorloopt:

zoeken: Je doet een gewone zoekoperatie op een binaire boom. Het pad is dus het pad van de wortel naar een top die de sleutel bevat of naar een top waar je niet verder kan gaan (als de top niet in de boom zit).

toevoegen: Je doet een gewone toevoegoperatie op een binaire boom. Het pad is dus het pad van de wortel naar het element dat toegevoegd moet worden. Het is dus ofwel het pad naar de nieuwe top (als het nog niet in de boom zat) of naar de top die het element bevat dat toegevoegd moest worden maar in feite al in de boom zat.

verwijderen: Je doet een gewone verwijderoperatie op een binaire boom. Het pad is dus het **hele** pad dat je in de boom doorloopt. De verwijderde top zit er natuurlijk niet meer in. Voorbeeld: in het geval dat de sleutel in een blad v zit, is het dus het pad van de wortel naar de ouder van v. **Belangrijk:** als de sleutel in een top met twee kinderen zit en dus een andere sleutel in de top geplaatst wordt, telt het pad naar deze andere sleutel mee!

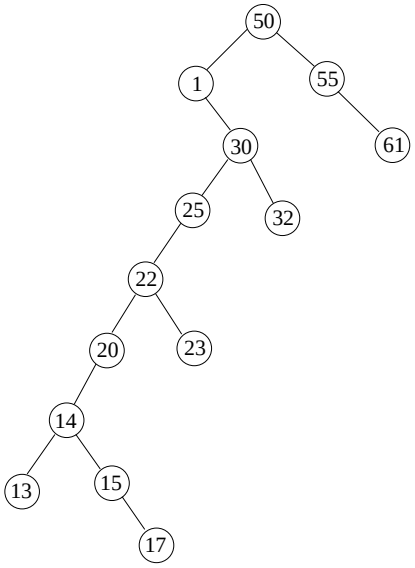
In gevallen van een zoekoperatie of een verwijderoperatie met een sleutel die niet in de boom zit of een toevoegoperatie van een sleutel die al in de boom zit, is het pad ook het hele pad dat je hebt doorlopen!



Figuur 6: Sleutel 7 wordt opgezocht en semi-splay langs het pad toegepast.

Een voorbeeld hoe dit werkt, kunnen jullie in Figuur 6 zien.
Als wij kijken wat er gebeurt als wij een sleutel opzoeken dan zien wij dat als het pad naar deze sleutel n bogen bevat voordat wij hem opzoeken, het pad van de wortel naar de sleutel achteraf nog ten hoogste $\lceil n/2 \rceil$ bogen bevat en de sleutel dus duidelijk dichter bij de wortel zit. Sleutels worden dus snel in de richting van de wortel verplaatst, maar als andere sleutels opgezocht worden, wordt de afstand alleen langzaam weer groter (zie oefening 7).
Sleutels die vaak opgezocht worden, gaan dus meestal dicht bij de wortel terechtkomen.

Oefening 6 *Voeg sleutel 18 op een semi-splay manier toe aan deze zoekboom en verwijder daarna sleutel 17.*



Oefening 7 *Stel dat je in een semi-splay boom semi-splay toepast op een pad van lengte n (dus $n+1$ toppen) en dat v een top in de boom is met afstand $m > 1$ van de wortel voordat je semi-splay toepast.*

Geef het resultaat van a.) tot d.) altijd als functie van m.

a.) *Stel dat v de top is die de sleutel bevat die je opzoekt. Wat is de grootst mogelijke en wat is de kleinst mogelijke afstand van v van de wortel nadat je semi-splay hebt toegepast?*

b.) *Stel dat v een arbitraire top is die op het pad ligt. Wat is de grootst mogelijke en wat is de kleinst mogelijke afstand van v van de wortel nadat je semi-splay hebt toegepast?*

c.) *Stel dat v een arbitraire top is die niet op het pad ligt. Wat is de grootst mogelijke afstand van v van de wortel nadat je semi-splay hebt toegepast?*

d.) *Stel dat v een arbitraire top is die niet op het pad ligt. Wat is de kleinst mogelijke afstand van v van de wortel nadat je semi-splay hebt toegepast?*

Hoeveel op elkaar volgende opzoekoperaties zijn ten hoogste nodig tot v in de wortel zit of een kind van de wortel is?

Toon aan dat jouw antwoorden juist zijn.

Definitie 4 *Een semi-splay boom is een binaire zoekboom T samen met de net gezien bewerkingen voor het opzoeken, verwijderen en toevoegen van toppen.*

Oefening 8 *Geef een binaire zoekboom die niet door middel van semi-splay toevoegbewerkingen alleen opgebouwd kan zijn.*

Een enkele bewerking op een semi-splay-boom kan lineaire kost hebben. Als je bv. de sleutels $1, 2, \ldots, n$ in deze volgorde toevoegt dan krijg je een boom die bijna helemaal een pad is. De afstand van 1 van de wortel is dan $n-2$. Als je daarna dus 1 opzoekt dan vraagt dat lineaire tijd. Maar in dit geval hadden wij eerst heel veel goedkope stappen zodat de gemiddelde tijd inderdaad niet lineair is maar constant. De gemiddelde tijd is natuurlijk niet altijd constant maar wel even goed als in gebalanceerde zoekbomen zoals in AVL-bomen:

Stelling 4 *Een reeks van n bewerkingen op een initieel lege semi-splay-boom vraagt ten hoogste $O(n \log n)$ stappen, dus gemiddeld $O(\log n)$ stappen per bewerking.*

Op dit moment kunnen wij deze stelling nog niet bewijzen, maar wij zullen later technieken zien waarmee wij dat wel kunnen.

Oefening 9 *De bedoeling van het verplaatsen van een opgezochte sleutel in een semi-splay-boom is duidelijk: je wil hem dichter bij de wortel hebben.*
Maar als je een element uit een semi-splay-boom verwijdert, wordt de ordening ook verstoord hoewel je dat misschien niet wil. Sleutels die misschien vrij vaak opgezocht worden, worden in de richting van de bladeren geschoven. Om sleutels die dicht bij de wortel zitten daar te houden, lijkt het verstandig bv. als je een blad verwijdert de boom helemaal niet te veranderen. Het nadeel is echter dat de performantiegarantieën dan niet meer geldig zijn.
Toon aan dat als je op deze manier sleutels verwijdert er reeksen van bewerkingen bestaan zodat n bewerkingen op een initieel lege semi-splay boom gemiddeld $\Theta(n)$ stappen per bewerking vragen (en de hele reeks dus $\Theta(n^2)$).

Oefening 10 *Wij weten dat een enkele bewerking op een semi-splay-boom een kost van $\Theta(n)$ kan hebben. Stelling 4 garandeert echter dat een reeks van n bewerkingen gemiddeld maar kost $O(\log n)$ kan hebben als je met een* **lege** *boom begint.*
Stel nu dat je niet met een lege boom begint maar dat je met een semi-splay-boom begint waarin al n sleutels zitten (die net met semi-splay zijn toegevoegd). Bewijs dan een van de volgende twee beweringen:

a.) *Als de boom al n sleutels bevat, kunnen de volgende n bewerkingen duidelijk duurder zijn dan $O(\log n)$ per bewerking. Precies: ze kunnen gemiddeld $\omega(\log n)$ stappen per bewerking vragen.*

b.) *Als de boom al n sleutels bevat dan vragen de volgende n bewerkingen nog altijd gemiddeld $O(\log n)$ stappen per bewerking.*

Oefening 11 *Wij hadden gezegd dat de gemiddelde tijd* natuurlijk niet altijd constant is *in een semi-splay boom.* Natuurlijk *betekent dat je dat gemakkelijk kan bewijzen. Dus:*
Stel dat het opzoeken van een sleutel in een zoekboom gebeurt door langs een pad van de wortel naar de top met deze sleutel te lopen. Toon aan dat er voor elke manier een binaire zoekboom te herbalanceren een reeks van bewerkingen bestaat die als je met een lege boom begint en n bewerkingen doet, gemiddeld $\Omega(\log n)$ toppen per bewerking bezoekt. Dat betekent natuurlijk dat het niet

mogelijk is gemiddeld een constante kost per bewerking te hebben, maar dat je ten minste $f(n) = c * \log n$ met een constante c als functie nodig hebt om een bovengrens voor het gemiddelde aantal stappen per bewerking te hebben.

2.3 Gebalanceerde zoekbomen

Natuurlijk zou het ideaal zijn als wij zouden kunnen garanderen dat zoekbomen altijd optimaal gebalanceerd zijn. Optimaal gebalanceerd betekent dat de diepte (de lengte (aantal bogen) van het langste pad van de wortel naar een blad) ten hoogste $\lfloor \log n \rfloor$ is. Wij hebben al een oefening gehad waarin wij gezien hebben dat dat in tijd $O(n)$ per bewerking inderdaad gegarandeerd kan worden, maar lineaire tijd is natuurlijk veel te duur. Jammer genoeg kan het niet beter:

Oefening 12 *Bewijs de volgende stelling:*

Stelling 5 *Als een manier om toppen aan een binaire boom toe te voegen garandeert dat de boom altijd optimaal gebalanceerd is, dus diepte $\lfloor \log n \rfloor$ heeft als er n sleutels inzitten, dan kan het gebeuren dat het herbalanceren na een toevoegbewerking niet in tijd $o(n)$ mogelijk is – om het even welk algoritme toegepast wordt.*

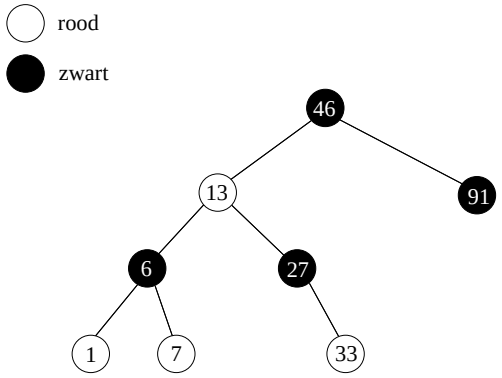
Tip: Kijk naar een perfect gebalanceerde boom met $2^n - 2$ elementen – bv. $1, 2, \ldots, 2^n - 2$. Voeg dan – afhankelijk van hoe de boom eruitziet – ofwel 0 ofwel $2^n - 1$ toe.
Bepaal (en bewijs) een ondergrens voor het aantal toppen waarbij de referentie naar de kinderen gewijzigd moet worden.

2.3.1 Rood-zwart bomen

Rood-zwart bomen zijn binaire zoekbomen waar – analoog met AVL-bomen – gegarandeerd is dat ze altijd goed gebalanceerd zijn. Maar de manier van doen is iets anders en wij moeten niet de diepte van de deelbomen bijhouden (maar wel andere informatie).

Definitie 5 *Een rood-zwart boom is een binaire zoekboom T waarvan de toppen ofwel rood ofwel zwart gekleurd zijn. Bovendien voldoet T aan de volgende eigenschappen:*

- *De wortel van T is zwart.*
- *De kinderen van een rode top zijn zwart*
- *Elk pad van de wortel naar een NULL-pointer bevat hetzelfde aantal zwarte toppen. Wij schrijven voor dit aantal $z(T)$ of gewoon z als duidelijk is om welke boom het gaat.*



Figuur 7: Een voorbeeld van een rood-zwart boom .

Hier is het niet onmiddellijk duidelijk dat de bomen goed gebalanceerd zijn, omdat niet expliciet eisen aan de diepte gesteld worden, maar de eis met *hetzelfde aantal zwarte toppen* lijkt er wel op. Het is ook niet onmidellijk duidelijk of je voor het herbalanceren de aantallen van zwarte toppen op de paden moet bijhouden (dat zou dan sterk op AVL-bomen lijken), maar wij zullen zien dat dat niet nodig is. Maar eerst bewijzen wij dat rood-zwart bomen inderdaad goed gebalanceerd zijn.

Lemma 6 *Als een pad van de wortel naar een NULL-pointer in een rood-zwart boom T precies z zwarte toppen bevat, bevat T ten minste $2^z - 1$ zwarte toppen en dus ook tenminste $2^z - 1$ toppen.*

Of anders gezegd: Voor een rood-zwart boom T geldt: T bevat ten minste $2^{z(T)} - 1$ zwarte toppen en dus ook tenminste $2^{z(T)} - 1$ toppen.

Bewijs: Inductie in $z = z(T)$:

Voor $z = 1$ is het duidelijk. Stel dus dat $z > 1$ is. Omdat de wortel w zwart is, zijn er twee kinderbomen van w (twee omdat er anders een pad met maar 1 zwarte top naar een NULL-pointer was). Kijk naar een pad van w naar een NULL-pointer in elk van de twee deelbomen en noem voor elk van de twee paden de tweede zwarte top die je tegenkomt (dus de eerste behalve w) w_1 resp. w_2. De toppen w_1 en w_2 zijn de wortels van twee deelbomen T_1, T_2 die ook rood-zwart bomen zijn (waarom?) en waarvoor geldt $z(T_1) = z(T_2) = z - 1$. Volgens inductie bevatten deze deelbomen dus ten minste $2^{z-1} - 1$ toppen. Als wij dus de toppen van de twee deelbomen en w tellen, is het resultaat dat de hele boom ten minste $2 * (2^{z-1} - 1) + 1 = 2^z - 1$ toppen bevat.

■

Oefening 13 *Gegeven een rood-zwart boom T. In Lemma 6 hebben wij een benedengrens voor het aantal toppen in T als functie van $z(T)$ bewezen.*

- *Toon aan dat dit een goede benedengrens is als alleen maar $z(T)$ van de boom gekend is.*
- *Bepaal bovengrenzen voor het aantal zwarte toppen en voor het aantal toppen in T als functie van $z(T)$.*
- *Toon aan dat jouw grenzen goede bovengrenzen zijn.*

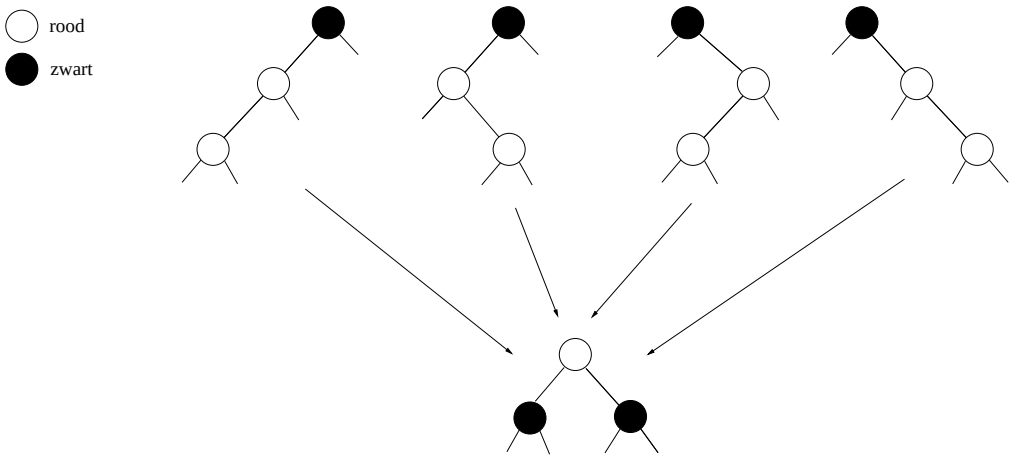
Corollarium 7 *In een rood-zwart boom met n toppen bevatten paden van de wortel naar een NULL-pointer ten hoogste* $\log(n+1)$ *zwarte toppen, de diepte is dus ten hoogste* $2\log(n+1)-1$.

Bewijs: Dat volgt direct met Lemma 6 en omdat de wortel zwart is. ■

In rood-zwart bomen is dus gegarandeerd dat je een top altijd in tijd $O(\log n)$ kan bereiken als er n toppen in de boom zitten. Opzoeken is dus geen probleem, maar wat doe je als je een top moet verwijderen of toevoegen? Dan wordt de boom gewijzigd en het is niet duidelijk of hij achteraf nog aan de eigenschappen van een rood-zwart boom voldoet.

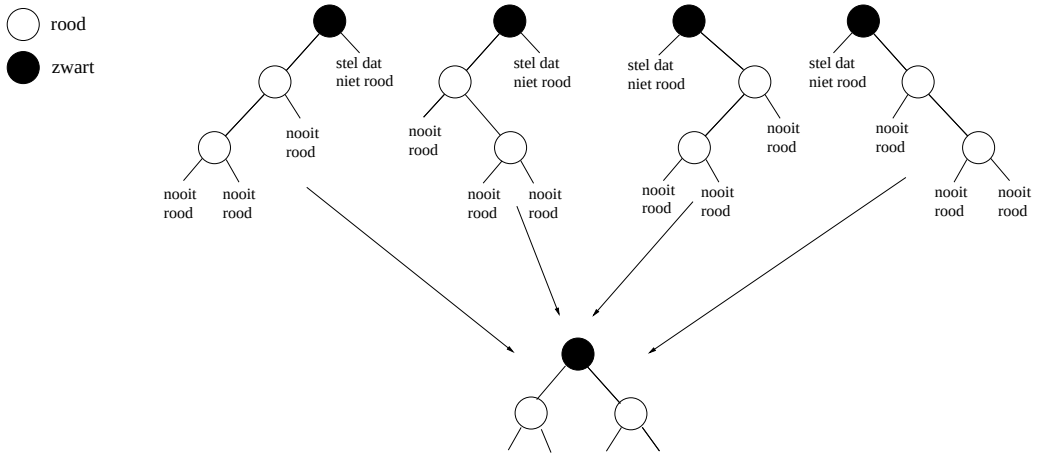
Toevoegen:
Als je in de rood-zwart boom in Figuur 7 bv. sleutel 77 toevoegt, kan je die gewoon rood kleuren en het is nog altijd een rood-zwart boom omdat het aantal zwarte toppen op de paden niet gewijzigd is en de ouder van de nieuwe top zwart is.
Als je sleutel 3 toevoegt heb je een probleem: je zou hem rood moeten kleuren om het aantal zwarte toppen op de paden niet te veranderen, maar dan heb je een rode ouder met een rood kind – en dat mag ook niet!
Behalve in het geval dat wij de eerste top toevoegen (die natuurlijk zwart moet zijn), voegen wij altijd rode toppen toe. Als er een probleem is en de boom niet meer aan de voorwaarden van een rood-zwart boom voldoet, gaan wij achteraf de boom in een bottom-up manier herstellen door kleuren en de structuur te wijzigen en *het probleem in de richting van de wortel te schuiven.* Stel dus dat wij een nieuwe rode top in een blad hebben toegevoegd. Als er een probleem is – dus de boom niet meer aan de voorwaarden voor een rood-zwart boom voldoet – kan het probleem alleen zijn dat de ouder al rood was. Omdat de ouder niet de wortel kan zijn (die is zwart) heeft de ouder zelf een zwarte ouder. Wij kunnen de deelbomen zo vervangen als in Figuur 8. Als wij dat doen, lossen wij één conflict op maar kunnen misschien (ten hoogste) één

nieuw conflict (dat is een nieuwe rode ouder met een rood kind) maken. Maar dit conflict is dan twee bogen dichter bij de wortel. Vroeger of later komen wij dus in de wortel terecht of kunnen stoppen omdat er geen nieuw conflict is. Maar ook al lijkt het slecht veel herbalanceringsoperaties te moeten doen, is het wel zo dat elke herbalanceringsoperatie een rode top minder en een zwarte top meer tot gevolg heeft. Toekomstige toevoegingen vragen dan vaak minder herbalanceringen omdat de kans op botsingen kleiner wordt. Herbalanceringen zijn dus ook zoiets als een investering in de toekomst...



Figuur 8: Het wijzigen van kleuren en kleine deelbomen in een rood-zwart boom .

Als wij in de wortel terechtkomen dan is het enige mogelijke probleem dat die nu rood is – dat mag niet. Maar omdat de wortel op **elk** pad van de wortel naar een NULL-pointer ligt, kunnen wij die gewoon zwart maken en daardoor $z(T)$ met 1 verhogen.
Als je naar de buitenbomen in Figuur 8 kijkt, dan zie je dat ten hoogste één van de buitenbomen – die van de bovenste top – een rode wortel kan hebben. Als dat niet zo is, **kan** je de boom ook op de manier in Figuur 9 wijzigen. Het voordeel is dat je dan gegarandeerd geen nieuw conflict hebt, dus onmiddellijk kan stoppen met herbalanceren. Nadeel is dat je dan ook een top moet bekijken die niet op het pad ligt. Je kan bewerkingen zoals in Figuur 8 ook als *investering in de toekomst* zien. Je verhoogt het aantal zwarte toppen in de boom en maakt daardoor de kans dat tijdens het toevoegen rode ouders rode kinderen krijgen – en de boom geherbalanceerd moet worden – kleiner.
Er zijn nog andere manieren om rood-zwart bomen efficiënter te implementeren (bv. al op weg naar de sleutel ervoor zorgen dat het herbalanceren vroeg moet stoppen door ervoor te zorgen dat twee zwarte toppen op elkaar



Figuur 9: Het wijzigen van kleuren en kleine deelbomen in een rood-zwart boom zo dat geen nieuw conflict kan ontstaan.

volgen), maar het principe is duidelijk. Omdat je met een constant aantal aanpassingen twee bogen dichter bij de wortel komt en het hele pad een lengte van $O(\log n)$ heeft, is duidelijk dat het herbalanceren in $O(\log n)$ tijd kan gebeuren.
Als je toppen moet verwijderen, doe je dat op dezelfde manier als in gewone binaire zoekbomen – je verwijdert dus nooit een top met twee kinderen maar vervangt die door een kind (als er maar 1 is) of je vervangt de sleutel erin door de grootste in zijn kleinerdeelboom of de kleinste in zijn groterdeelboom en de top van de verplaatste sleutel wordt verwijderd. Daarbij zijn de rode toppen niet het probleem – als je die verwijdert blijft het een rood-zwart boom. Maar als je een zwarte top verwijdert – bv. een zwart blad – is het aantal zwarte toppen niet meer hetzelfde voor alle paden van de wortel naar een NULL-pointer. Ook dat kan alleen door lokale aanpassingen van kleine deelboompjes langs het pad hersteld worden. Maar omdat je verschillende gevallen hebt die niet echt nieuwe ideeën geven, gaan wij dat hier niet uitwerken.

Oefening 14 *Wat is er mis met het bewijs van de volgende bewering:*
Bewering: *Elke rood-zwart boom met ten minste 2 toppen bevat ten minste één rode top.*

Argumentatie: Wij bewijzen het met inductie. Als er 2 toppen zijn dan is dat altijd een zwarte ouder met een rood kind. De bewering klopt dus.
Voor de inductie gebruiken wij de geoptimaliseerde toevoegoperaties. Een rood-zwart boom met $n + 1 > 2$ toppen is ontstaan uit een rood-zwart boom

met n toppen. Als na het toevoegen niet geherbalanceerd moet worden, is de nieuwe top rood, dus voldoet de boom aan de bewering. Stel dus dat geherbalanceerd wordt. Als de geoptimaliseerde bewerkingen toegepast kunnen worden, heb je twee rode kinderen. Die zijn natuurlijk niet de wortel, blijven dus rood en de boom voldoet aan de bewering – om het even of nog meer gebalanceerd moet worden. Als de geoptimaliseerde bewerkingen **niet** *toegepast kunnen worden, betekent dat dat ook het tweede kind van de bovenste (zwarte) top die bij de herbalancering betrokken is rood is. Dit kind wordt na het balanceren een kind van de nieuwe deelboom en blijft rood – om het even of nog meer gebalanceerd moet worden. Ook in dit geval zit er dus een rode top in de boom en de bewering is bewezen.*

Oefening 15 *Voeg de sleutels 10,5,2,20,1,7,9,30 in deze volgorde toe aan een rood-zwart-boom.*

Toon voldoende tussenstappen om te zien wat er gebeurt.

Oefening 16 *Lemma 6 geeft een benedengrens voor het aantal toppen in een rood-zwart-boom als er b zwarte toppen op een pad van de wortel naar een NULL-pointer zitten.*

- *Kan je ook een benedengrens voor het aantal toppen bewijzen als er r rode toppen op een pad van de wortel naar een NULL-pointer in een rood-zwart-boom zitten?*
- *Kan je een bovengrens voor het aantal toppen bewijzen als er r rode toppen op een pad van de wortel naar een NULL-pointer in een rood-zwart-boom zitten?*

Oefening 17 *Stel dat je een zwart blad moet verwijderen in een rood-zwart boom en dat de ouder van dit blad rood is. Werk uit op welke manier je deelboompjes moet vervangen om de balanceringsvoorwaarden te herstellen.*

2.3.2 2-3 bomen

Nu zullen wij zoekbomen zien die geen binaire bomen zijn: 2-3 bomen. Je zou misschien verwachten dat 2-3 over het aantal sleutels gaat, maar het *binair* in *binaire bomen* gaat ook over het aantal kinderen. Ook hier betekent 2-3 boom dat het aantal kinderen 2 of 3 is. Maar er zijn nog meer voorwaarden:

Definitie 6 *Een 2-3 boom is een zoekboom met de volgende eigenschappen:*

- *Elke top bevat ofwel 1 ofwel 2 sleutels.*

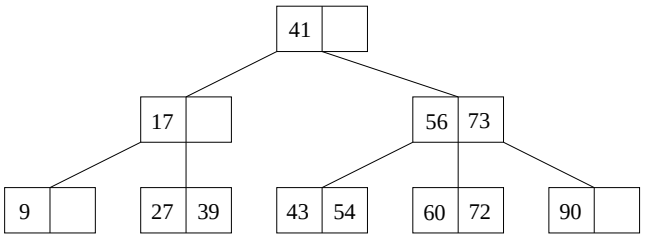
- *Een top met i sleutels heeft ofwel $i+1$ lege kinderen (dan is hij een blad) ofwel $i+1$ niet lege kinderen. Toppen met lege kinderen* **en** *niet lege kinderen bestaan niet.*

- *Alle bladeren zitten op dezelfde diepte.*

Oefening 18 *Is de volgende definitie equivalent?*

Een 2-3 boom is een zoekboom met één of twee sleutels in elke top waar voor elke top t geldt dat alle kinderen van t dezelfde afstand tot een NULL-pointer hebben. Hierbij tellen wij de afstand van een leeg kind tot een NULL-pointer als 0*.*

Op voorhand is het zelfs niet duidelijk dat dergelijke bomen voor elke verzameling van sleutels bestaan. Als bv. alle toppen één sleutel bevatten, dan bestaan dergelijke bomen alleen voor aantallen van sleutels die gelijk zijn aan $2^k - 1$ voor een $k \in \mathbb{N}, k > 0$. (Waarom?) Maar wij zullen zien dat je aan een gegeven boom een top kan toevoegen en de boom herbalanceren. Omdat een boom met maar één top aan de definitie voldoet, volgt met inductie dat dergelijke bomen voor alle verzamelingen van sleutels bestaan.



Figuur 10: Een voorbeeld van een 2-3 boom.

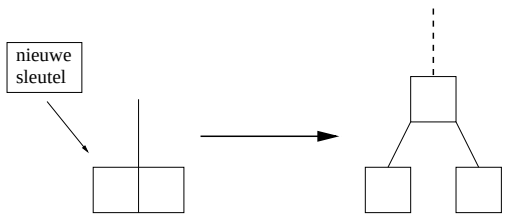
Het is duidelijk dat als wij de diepte als maatstaf nemen 2-3 bomen fantastisch presteren:

Lemma 8 *Een 2-3 boom met diepte k bevat ten minste $2^{k+1} - 1$ sleutels, de diepte van een 2-3 boom met n sleutels is dus ten hoogste $\lfloor \log(n+1) \rfloor - 1 \leq \lfloor \log n \rfloor$.*
De bovengrens voor de diepte van een 2-3 boom is dus ten hoogste even groot als de benedengrens voor de diepte van gelijk welke binaire zoekboom.

Maar natuurlijk is de diepte niet noodzakelijk een goede maatstaf! In 2-3 bomen heb je meer sleutels per top dus moet je bv. in het slechtste geval meer vergelijkingen doen om uit te vissen waar je je sleutel moet zoeken. Gebalanceerde binaire zoekbomen en 2-3 bomen hebben allemaal een kost

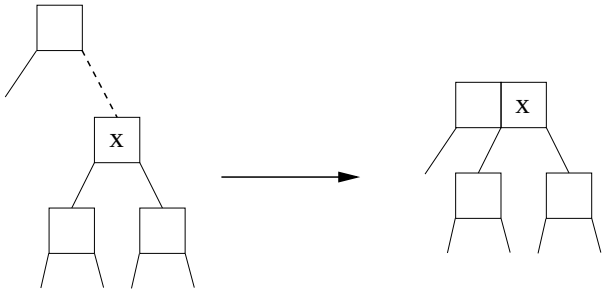
van $O(\log n)$ per bewerking. Terwijl het aantal toppen dat je moet bezoeken een constante factor groter kan zijn als je met binaire bomen werkt, is de kost per top een constante factor kleiner voor binaire bomen. Welke factor belangrijker is, hangt dan van de precieze toepassing af. In DA III zullen jullie bomen zien met veel meer sleutels per top die vooral in speciale toepassingen goed presteren!
Maar hoe kan je een 2-3 boom herbalanceren als de balans door het toevoegen of verwijderen verstoord wordt?

Toevoegen:
Nieuwe toppen worden nooit in het midden van de boom toegevoegd maar altijd in een blad. Zelfs als er ruimte in het midden van een 2-3 boom zou zijn (een top met maar één sleutel), zou je niet alleen de 2-3 boom eigenschappen kunnen verstoren, maar om te kijken of het tenminste een zoekboom zou zijn, zou je toch tot een blad moeten lopen (waarom?).
Het blad waar hij toegevoegd moet worden, zoeken wij op de gewone manier voor een zoekboom. Als dit blad tot nu toe maar één sleutel bevat, is er geen probleem. Anders moet de boom geherbalanceerd worden. Wij zullen het herbalanceren zo beschrijven dat wij tijdelijk één deelboom toelaten die op zich ook een 2-3 boom is, maar waar alle bladeren op een diepte zitten die één groter is dan de bladeren die niet in de deelboom zitten. Dat noemen wij een bijna-2-3 boom. De wortel van deze bijna-2-3 boom zal altijd maar één sleutel bevatten.
Stel dus dat wij de sleutel in een blad moeten plaatsen waarin er al twee sleutels zitten. Dan maken wij van de ene top met twee sleutels een complete binaire boom met diepte 1. Die heeft voldoende ruimte voor 3 sleutels. Jammer genoeg zitten de 2 bladeren van deze deelboom één dieper dan de andere bladeren – het resultaat is dus alleen een bijna-2-3 boom en geen 2-3 boom. Deze operatie zie je in Figuur 11. De verdeling van x,y,z op de toppen in het rechterdeel hangt er natuurlijk vanaf hoe groot de nieuwe top in vergelijking met de anderen is. Een boog naar de wortel van de deelboom die ervoor zorgt dat de boom maar een bijna-2-3 boom is, wordt met puntjes getekent. Een bijna-2-3 boom heeft dus precies één zo'n boog.



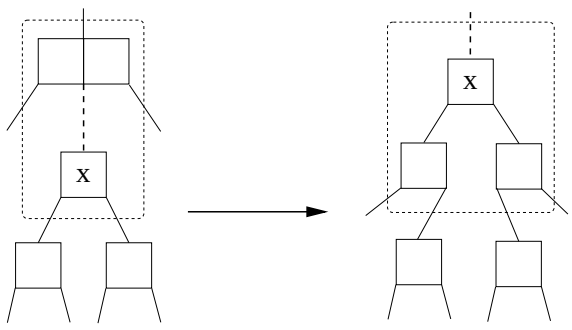
Figuur 11: Het vervangen van een blad door een kleine binaire boom.

Nu moeten wij de wortel van de deelboom in de richting van de wortel van de hele boom schuiven. Als op weg naar de wortel de diepte van de deelboom 1 kleiner gemaakt kan worden, is de bijna-2-3 boom terug een 2-3 boom en we zijn klaar. Dat is het geval als de ouder van de wortel maar één sleutel bevat. Hoe je dat kan doen zie je voor het geval dat de deelboom een rechterdeelboom is in Figuur 12.

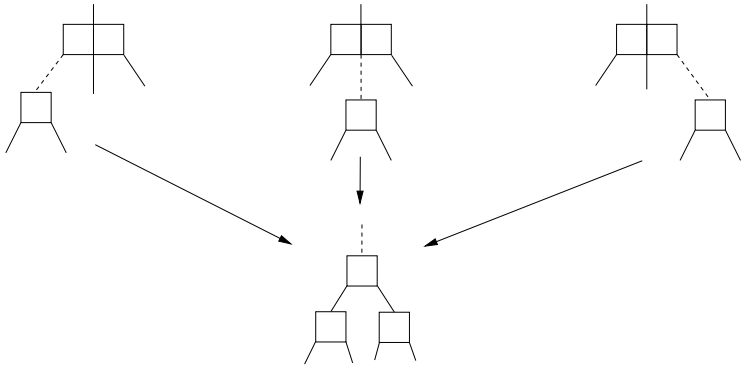


Figuur 12: Een bijna-2-3 boom wordt terug een 2-3 boom als de ouder van de wortel van de deelboom die te diep is maar één sleutel bevat. Hier het geval dat de deelboom met de te grote diepte een rechterkind is.

Anders – dus als de ouder al 2 sleutels bevat – gaan wij gewoon door. De wortel van de foutieve deelboom zal achteraf één stap dichter bij de wortel zijn. Deze operatie is het vervangen van een ouder met twee toppen en een kind met 1 top door een complete binaire boom met 3 toppen. Jullie zien dat voor het geval dat de deelboom een midden-deelboom is in Figuur 13. De operatie lijkt in feite sterk op het plaatsen van de sleutel in een blad met twee sleutels. . . Let ook hier goed op of de diepten allemaal zijn zoals het voor een bijna-2-3 boom nodig is!



Figuur 13: Een bijna-2-3 boom wordt gewijzigd zodat de wortel van de deelboom met een te grote diepte één niveau hoger zit. Dit is het geval waar de deelboom waar de diepten niet goed zijn een middenkind is.

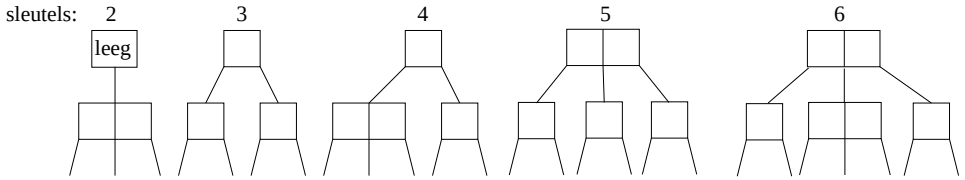


Figuur 14: Alle drie mogelijke gevallen hoe de deelboom met de foutieve diepten kan zitten en hoe de bijna-2-3 boom wordt gewijzigd.

Als wij de wortel bereiken is de deelboom de hele boom, dus hebben **alle** bladeren een diepte die 1 groter is – de bijna-2-3 boom is dus terug een 2-3 boom.

Verwijderen:
Het verwijderen (zonder herbalanceren) lijkt op de manier waarop je het zou doen in een binaire zoekboom, alleen dat hier natuurlijk het geval niet kan opduiken dat je een top met maar één kind hebt. Als je een sleutel uit een top verwijdert die geen blad is, heeft de sleutel altijd een rechter **en** een linkerkind. Let op: links of rechts van de sleutel kan het middenkind van de top zitten – met *links van de sleutel* bedoelen wij het kind met dezelfde index als de sleutel en met *rechts van de sleutel* bedoelen wij het kind met een index die 1 groter is. Je kan de sleutel dus bv. gewoon vervangen door de grootste sleutel in het linkerkind of de kleinste sleutel in het rechterkind. Deze sleutels zitten altijd in bladeren (waarom?) waaruit ze verwijderd moeten worden. Het komt er dus altijd op neer dat je een sleutel uit een blad moet verwijderen.
Als de sleutel uit een blad wordt verwijderd waarin er twee sleutels zitten, zijn de voorwaarden van een 2-3 boom niet geschonden. Dus stel dat er maar één sleutel zit. Wij werken opnieuw met een 2-3 boom die op één plaats een fout heeft. Je kan de situatie op meerdere manieren voorstellen: als een foutieve 2-3 boom met één deelboom waarin de diepte van de bladeren om 1 te klein is (analoog met het toevoegen) of als een foutieve 2-3 boom met één top die leeg is en één kind heeft. Wij kiezen hier voor de tweede manier om het te beschrijven.
Er is dus een top zonder sleutel en met één kind (dat in het begin als deze top een blad is, leeg is). Wij noemen dat opnieuw een bijna-2-3 boom. Dan verschuiven wij deze foutieve top naar boven tot de fout verdwijnt of hersteld

kan worden. Wij zullen niet alle gevallen expliciet tonen maar gewoon aantonen dat het door middel van lokale operaties hersteld kan worden. Voor deze operaties hebben wij alleen de ouder en zijn kinderen nodig. Als wij kijken hoeveel sleutels in de ouder en de broers van de lege top samen zitten dan zijn dat ten minste 2 en ten hoogste 6. In Figuur 15 zie je bomen die je kan gebruiken om de ouder van de foutieve top en zijn kinderen te vervangen om ervoor te zorgen dat de boom aan de voorwaarden van een 2-3 boom voldoet. Je vervangt een deelboom waar alle buitenbomen aan een top op diepte 1 hangen door een andere boom met diepte 1 waar de buitenbomen aan een top met diepte 1 hangen. Dat zorgt ervoor dat geen nieuwe voorwaarden voor een 2-3 boom geschonden worden.



Figuur 15: Vervangbomen voor het herbalanceren van een 2-3 boom na het verwijderen van een sleutel.

Je ziet dat in bijna alle gevallen de foutieve top onmidellijk verdwenen is – de boom dus terug een 2-3 boom is. En in het geval dat er maar 2 toppen in de ouder en zijn kinderen zitten, is de nieuwe positie van de foutieve top één stap dichter bij de wortel. Wij kunnen dus doorgaan tot de foutieve top in de wortel zit of de fout verdwenen is. Als de top in de wortel zit kunnen wij hem gewoon verwijderen en zijn kind de nieuwe wortel maken. Het resultaat is opnieuw een 2-3 boom.
Deze operaties tonen dat het **mogelijk** is alleen met lokale operaties – dus in tijd $O(\log n)$ de boom te herstellen. Als je het echt efficiënt wil doen – dus als de constanten ook belangrijk zijn – moet je natuurlijk zorgvuldig kiezen hoe je het het best implementeert.

Oefening 19 *Bij het herbalanceren van een 2-3 boom na een toevoegbewerking houden wij altijd alleen rekening met 2 toppen – de wortel van de foutieve deelboom en zijn ouder. Beschrijf herbalanceringsbewerkingen die ook rekening houden met de kinderen van deze toppen en het toelaten in veel gevallen al vroeger met het herbalanceren te stoppen.*

Oefening 20 *(a) Geef een definitie van een 2-3-4-boom die analoog is aan die van een 2-3 boom.*

(b) Definieer een toevoegbewerking en toon aan dat je de boom in tijd $O(\log n)$ kan herbalanceren als hij n toppen bevat en na de toevoegbewerking niet meer aan de voorwaarden van de definitie voldoet.

(c) Definieer een verwijderbewerking en toon aan dat je de boom in tijd $O(\log n)$ kan herbalanceren als hij n toppen bevat en na de bewerking niet meer aan de voorwaarden van de definitie voldoet.

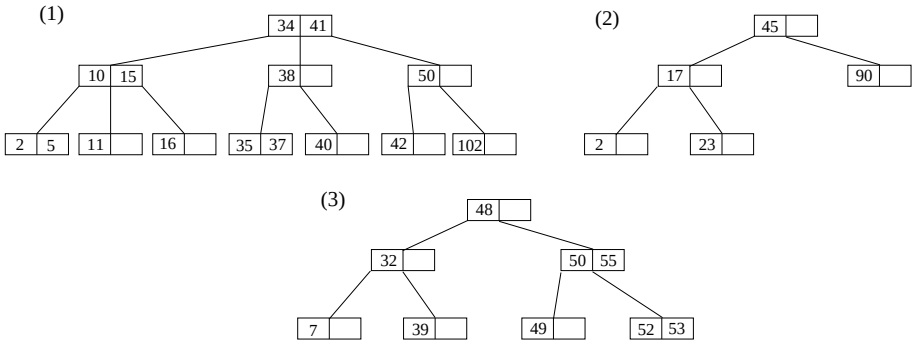
(d) Vergelijk deze 2-3-4-boom met een rood-zwart boom . Interpreteer rood-zwart bomen als een manier om 2-3-4-bomen voor te stellen. Tip: interpreteer een zwarte top en zijn rode kinderen als één top. Zou deze boom aan de definitie van een 2-3-4-boom voldoen?

Oefening 21 *Dit is een oefening uit een examen:*

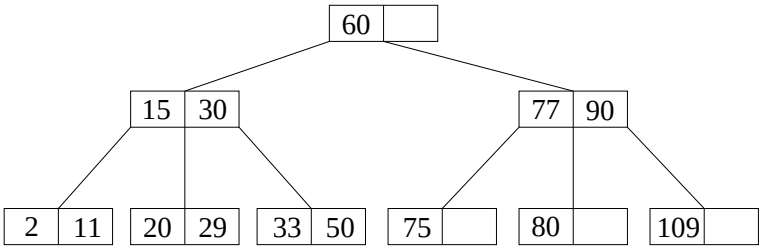
a.) *Wat is in het slechtste geval de diepte van een 2-3-boom met n sleutels en wat is de diepte in het beste geval ? Wat zijn de beste en slechtste gevallen ? (Geen bewijs vereist.)*

b.) *Hoeveel vergelijkingen van sleutels zijn er in het beste geval nodig om een nieuwe sleutel aan een 2-3-boom met n sleutels toe te voegen ? Wat is het beste geval ? (Geen bewijs vereist.)*

c.) *Schrijf voor elk van de drie bomen of hij een 2-3-boom is en in het geval van niet waarom niet.*

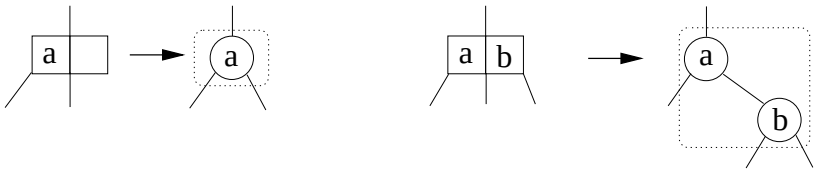


d.) *Voeg de sleutel* 24 *toe aan de volgende 2-3-boom. Toon voldoende tussenstappen om te zien wat er gebeurt, maar voor de tussenstappen moet je niet elke keer de* **hele** *boom tekenen.*



Oefening 22 *Als je ook meerdere sleutels met dezelfde waarde in een boom wil opslaan moet je tenminste sommige van de "<" in de definitie van een zoekboom veranderen in "≤". Natuurlijk wil je zo weinig mogelijk van de "<" wijzigen om beter te weten waar de gelijke sleutels zitten. Als je zo'n algemenere 2-3 boom wil hebben, op welke plaatsen moet je de "<" ten minste wijzigen om ervoor te zorgen dat voor alle (eindige) rijen van mogelijke sleutels een 2-3 boom met deze sleutels bestaat.*

Oefening 23 *Je kan een 2-3-boom* **intern** *in principe ook voorstellen als een binaire boom, als je de toppen van de 2-3-boom voorstelt zoals in de volgende figuur:*



Stel dat de boom n sleutels bevat. Geef een **goede** *bovengrens voor de maximale afstand van een blad van de wortel in de te gronde liggende binaire boom en een goede benedengrens voor de minimale afstand. De $O()$-notatie is* **niet** *voldoende.*

3 Geamortiseerde complexiteit en technieken om hem te berekenen

Soms is het niet zo belangrijk hoe duur een enkele toepassing van een algoritme of een enkele bewerking op een datastructuur is, maar hoe duur een hele reeks van bewerkingen is. In een bank is het bv. soms belangrijk dat een zekere reeks van updates van de databank om middernacht gedaan is. Het is niet belangrijk hoe lang een enkele van de bewerkingen duurt, maar **heel** belangrijk hoe lang de hele reeks van bewerkingen van de dag duurt.
In het volgende zullen wij sommige voorbeelden zien waar het slechtste geval van een enkele bewerking *relatief* duur kan zijn, maar waar een hele reeks van bewerkingen gemiddeld toch heel goed presteert. We doen dus nog altijd een *slechtste geval analyse* maar nu gemiddeld over een hele reeks bewerkingen. Dat noemen wij de geamortiseerde complexiteit van een bewerking in de reeks.
Inderdaad is het niet echt iets nieuws naar hele reeksen van bewerkingen te kijken: je kan ook een toevoegoperatie op een boom als een reeks van operaties zien (vergelijken, herbalanceren,...) en je kan bv. een reeks van n toevoegoperaties op een initieel lege boom ook als één *opbouwoperatie* zien. Over wat een reeks van bewerkingen is en wat een enkele bewerking is, kunnen de meningen dus verschillen.
De verschillende technieken die jullie hier zien om deze complexiteit te bepalen, zijn gewoon rekentrucjes die het in veel gevallen gemakkelijker maken iets te berekenen dat zonder deze technieken echt moeilijk zou kunnen zijn. De technieken zijn **niet het doel** – het doel is de geamortiseerde complexiteit te bepalen en als je voor een probleem op een gemakkelijke manier een goede bovengrens kan bepalen, moet je deze technieken natuurlijk niet toepassen.
Jullie kennen zeker het verhaal van de jonge Gauss die alle getallen van 1 tot 100 moest samentellen en het idee had om ze gewoon zo te ordenen dat je 50 paren hebt waar de som van elk van de paren 101 was ($1 + 100$, $2 + 99$, ...). (Ik heb geen idee of dat ooit echt gebeurd is.) Dit is een gemakkelijk voorbeeld dat toont hoe een zekere manier om iets te berekenen kan helpen het sneller of gemakkelijker te doen.
Een ander voorbeeld waar een wiskundig werktuig niet het doel maar wel heel nuttig is, is als je wilt weten wat precies het minimum van (bv.) $3x^2 - 7x + 5$ is. Als je functies niet zou kunnen afleiden, zou het lastig zijn het minimum precies te bepalen. Het afleiden is hier een nuttig werktuig om jouw doel te bereiken – ook al heeft het doel zelf met afleiden niets te maken.

Maar ook al zijn het *rekentrucjes*: in dit deel gaat het niet om wiskunde maar om de complexiteit van algoritmen en datastructuren en hoe je die kan

bepalen. En dat is duidelijk een deel van de informatica...

Definitie 7 *De geamortiseerde kost van een bewerking in een reeks van bewerkingen is de hoogst mogelijke kost van de reeks gedeeld door het aantal bewerkingen.*
Om te beklemtonen dat wij het over een reeks van bewerkingen hebben, noemen wij de hoogst mogelijke kost van een gegeven reeks bewerkingen op een datastructuur soms ook de geamortiseerde kost van deze reeks bewerkingen – maar voor een reeks bewerkingen is dat natuurlijk hetzelfde als de gewone *kost die je krijgt als je de reeks als één grote bewerking beschouwt.*

Als de reeks m bewerkingen bevat en de duurste bewerking heeft kost $O(f)$ voor een functie f, dan is $O(m \cdot f)$ natuurlijk een bovengrens voor de geamortiseerde kost van de reeks, maar wij zullen zien dat wij in sommige gevallen een veel betere bovengrens kunnen bewijzen.
Natuurlijk zouden ook de *verwachte* kosten interessant zijn, maar dat is nog moeilijker. Daarvoor zou je inderdaad ook moeten weten hoe de mogelijke inputs van een algoritme verdeeld zijn en dat kan natuurlijk heel sterk verschillen. Een voorbeeld zijn de semi-splay bomen: stel dat je eerst een zeker aantal sleutels toevoegt en dan alleen maar sleutels opzoekt. Als de verdeling van de opgezochte sleutels zo is dat sommige sleutels heel vaak opgezocht worden en anderen duidelijk minder, is de verwachte kost veel minder dan in het geval dat de verdeling zo is dat alle sleutels even vaak opgezocht worden. Hier zit het probleem dus niet alleen bij de moeilijkheid het te berekenen, maar al vroeger: het is moeilijk te weten wat de juiste verdeling is.

We zullen alle methoden tonen door ze op dezelfde voorbeelden toe te passen zodat je de verschillen goed kan zien.

Voorbeeld 1 Een binaire teller verhogen:
Eerst bediscussiëren wij een voorbeeld dat op het eerste gezicht niet echt belangrijk lijkt, maar later zullen wij een datastructuur behandelen waar wij precies dit voorbeeld nodig hebben: het incrementeren van een binaire teller. Je hebt een rij van n booleaanse variabelen $x_{n-1}, \ldots, x_0$ waar je de binaire voorstelling van een getal wil opslaan. Als het getal 3 is en $n = 5$ zou de rij dus $0 \quad 0 \quad 0 \quad 1 \quad 1$ zijn. Het grootste getal dat je kan voorstellen is $2^n - 1$. Het wijzigen van de inhoud van een variabele van 0 naar 1 of van 1 naar 0 heeft kost 1. Als een getal gegeven is en je wil er 1 bijtellen moet je dus vanaf x_0 alle variabelen die 1 zijn, wijzigen tot 0 tot je de eerste variabele x_i (die met de kleinste index) vindt die waarde 0 heeft of tot je naar alle variabelen hebt gekeken en weet dat ze allemaal 1 zijn (dan krijg je een overflow). Dan

wijzig je x_i tot 1 *en alle bezochte variabelen met waarde* 1 *tot* 0. *Als je daarbij naar j variabelen moet kijken is de kost dus j.*

De bewerking is dus

- *één bijtellen*

en wij willen weten wat de geamortiseerde kost van een reeks is die begint met de waarde 0 *(dus waar in het begin alle variabelen* 0 *zijn) en waar dan 2^n keer* 1 *bijgeteld wordt. De bewerkingen stoppen dus als eerst alle variabelen* 1 *zijn en dan alles terug op* 0 *wordt gezet.*

Het tweede voorbeeld is duidelijk nuttig:

Voorbeeld 2 Een array groter maken:
Als je met een array werkt, heeft die altijd een vaste grootte. Je kan hem ook niet gewoon langer maken omdat het mogelijk is dat er in het geheugen andere variabelen volgen die je niet gewoon kan overschrijven. Je moet dus ergens anders meer plaats alloceren en de inhoud van de array kopiëren. Wij houden hier geen rekening met de manier waarop dat in besturingssystemen met virtueel geheugen echt wordt gedaan, maar werken met ons model – ten slotte is het maar een voorbeeld dat alleen bedoeld is om de technieken duidelijk te maken.
Een bekende manier om dat te doen is dat je elke keer dat je meer elementen wil toevoegen dan in de array passen, dubbel zo veel ruimte voor de array alloceert en de oude array kopieert.
Hoe duur dat is, hangt er ook van af, of je de niet-gebruikte elementen op 0 *wilt initialiseren of niet. Als je dat niet doet , is de kost de kost van het kopiëren, dus $O(g)$ waarbij g de grootte van de kleinere array is, maar anders is de kost $O(G)$ met G de grootte van de grotere array. Als de grote array 2 keer zo groot is als de kleine verschillen de kosten natuurlijk maar om een constante – als je alleen naar de $O()$-notatie kijkt, is er in dat geval dus geen verschil. Hier stellen wij dat de kost van zo'n kopieerbewerking het aantal elementen is dat gekopieerd moet worden.*
Wij werken nu met een stapel die als array geïmplementeerd is. De bewerkingen zijn

- *een element op het einde van de array toevoegen*

- *het laatste element in de array verwijderen*

De kost van het verwijderen is altijd 1 *en de kost van het toevoegen is* 1 *als geen nieuwe array aangemaakt moet worden en $1 + g$ waarbij g het aantal elementen is dat gekopieerd moet worden als de array uitgebreid moet worden.*

Wij willen weten wat de geamortiseerde kost van een reeks van n bewerkingen is als je van een lege array met maar 1 element vertrekt.

Het slechtste geval maal aantal bewerkingen

Als wij voor de binaire teller naar het slechtste geval voor één bewerking kijken dan is dat duidelijk een kost van n: als de waarde bv. $2^{n-1}-1$ is dan is x_{n-1} de enige 0 – alle n variabelen moeten dus gewijzigd worden. Als wij dat gebruiken om de geamortiseerde kost te berekenen dan hebben wij een kost van $n \cdot 2^n$ of een kost van n per bewerking.

Als wij naar het slechtste geval voor de array kijken, is dat natuurlijk als er een kopieerbewerking gedaan moet worden. Als het k-de element toegevoegd wordt dan is de kost $k-1+1$, de som dus $\sum_{k=1}^{n} k = n(n+1)/2$ en is $O(n^2)$ een bovengrens voor de geamortiseerde kost van de reeks (dus $O(n)$ per bewerking).
Tot nu toe hebben wij nog niet gebruikt dat wij de grootte van de array elke keer verdubbelen. Deze grens werkt dus voor elke manier om de array groter te maken – zelfs als er maar een enkele nieuwe plaats toegevoegd wordt.

In het geval van de binaire teller is de dure bewerking een *gewone* bewerking maar hier is het al duidelijk dat je in principe de dure uitbreidbewerking van de goedkopere bewerkingen kan splitsen en die dan apart kan tellen. Dat is al een eerste stap om het iets beter te doen dan als kost gewoon *het slechtste geval maal het aantal bewerkingen* te nemen. Omdat de dure bewerkingen maar $\log n$ keren kunnen gebeuren, zouden wij een geamortiseerde kost van $O(n \log n)$ voor de reeks van bewerkingen krijgen.

Belangrijk: Als deze gemakkelijke analyse al een goede bovengrens geeft – dus een bovengrens waarvoor je een reeks van bewerkingen kan vinden die echt zoveel stappen nodig heeft – dan hoeven natuurlijk geen ingewikkeldere technieken toegepast te worden. Probeer de complexiteit van de reeks van bewerkingen altijd met de meest eenvoudige middelen te bepalen en gebruik de ingewikkeldere technieken alleen als de gemakkelijke technieken geen goede grenzen opleveren.

Maar voor de voorbeelden (en ook voor semi-splay) zullen jullie dergelijke reeksen niet vinden...

Nu zullen wij drie technieken zien waarmee je soms een betere schatting kan maken dan gewoon het slechtste geval per bewerking te nemen. Natuurlijk moet je als je een bovengrens voor iets bepaalt, er ook over nadenken of dat een goede bovengrens is. De bovengrenzen die wij tot nu toe voor de voorbeelden hebben bepaald zijn bv. geen goede bovengrenzen. Maar wij

zullen bewijzen dat de gemiddelde kost per bewerking constant is en dan hoef je natuurlijk niet meer te bewijzen dat dat een goede grens is – dat is optimaal!
Geen van de technieken kan garanderen dat ze een betere bovengrens oplevert dan als je het slechtste geval neemt voor elke bewerking. Het kan zelfs gebeuren dat er geen betere bovengrens **is**. Het zijn gewoon technieken waarvoor je kan proberen of je ze kan toepassen en soms lukt het en soms niet. Maar als je ze niet kent, lukt het zeker nooit...

3.1 De aggregaatmethode

Het principe van de aggregaatmethode kan je als volgt beschrijven:

- Splits de bewerkingen in heel kleine elementaire stappen die gemakkelijk zijn om te tellen – waarvoor het dus gemakkelijk is om te zien hoe vaak ze gebeuren.
- Probeer dan deze elementaire stappen – die typisch een constante kost hebben – te tellen.

Als een complexe bewerking samengezet is uit elementaire bewerkingen, dan wordt de kost van deze complexe bewerking bepaalt door de som van de kosten van de elementaire bewerkingen. Als de elementaire stappen toch niet zo elementair zijn en geen constante kost hebben, dan moet natuurlijk de som van alle kosten van elementaire stappen geteld worden en niet het aantal elementaire stappen.

Toegepast op voorbeeld 1 – de binaire teller:
Hier kiezen wij de elementaire stappen als

(i) de inhoud van een variabele wijzigen – dat noemen wij een *bitflip*.

Een bitflip is dus het wijzigen van een variabele van 0 naar 1 of van 1 naar 0. Bij de berekening van de kost van een opteloperatie hebben wij dat al als kost 1 gerekend. Hoeveel van deze bitflips kunnen gebeuren?

De variabele x_0 wordt in elke stap gewijzigd – dus zijn er 2^n bitflips. Als wij naar variabele x_i voor $i > 0$ kijken dan wordt hij alleen maar gewijzigd als alle x_j met $j < i$ gelijk zijn aan 1. Alle 2^i mogelijke verdelingen van 0 en 1 op de variabelen $x_0, \ldots x_{i-1}$ duiken even vaak op (omdat ze alle getallen $0, \ldots, 2^i - 1$ voorstellen) dus worden op variabele x_i precies $2^n/2^i$ bitflips toegepast. En inderdaad geldt deze formule ook voor $i = 0$ omdat $2^n = 2^n/2^0$.

Als wij dat nu samentellen krijgen wij
$\sum_{i=0}^{n-1} 2^n/2^i = 2^n \sum_{i=0}^{n-1} 1/2^i < 2^n * 2$.
Dus was onze bovengrens voor de geamortiseerde complexiteit van $n \cdot 2^n$ of $O(n)$ per bewerking helemaal niet goed: Wij hebben net een bovengrens van $2 \cdot 2^n$ bewezen – dus gemiddeld een constante kost van $O(1)$ per bewerking.

Toegepast op voorbeeld 2 – een array groter maken:
Hier kiezen wij de elementaire stappen als

(i) een element verwijderen

(ii) een element voor de eerste keer in de lijst schrijven

(iii) een element kopiëren

Elke van deze stappen heeft een constante kost van 1 – dat hebben wij al bij het bepalen van de kost van de bewerkingen gebruikt – wij moeten nu nog tellen hoe vaak deze elementaire stappen kunnen gebeuren.
Als er n operaties zijn, moeten de stappen (i) en (ii) samen precies n keer gedaan worden – dat heeft dus een kost van n. Dat zou zelfs dan gelden als wij niet met een lege array zouden beginnen.
Op het einde kunnen er ten hoogste n elementen in de array zitten. (Hier gebruiken wij wel dat wij met een lege array beginnen.) Voor $n = 1$ is het duidelijk dat je maar één stap nodig hebt, stel dus dat $n \geq 2$. Hoe vaak werd de array dan groter gemaakt? Als hij x keer groter wordt gemaakt, zijn er 2^x plaatsen in de array. Als $2^{x-1} \geq n$ zou hij al na $x - 1$ keer groot genoeg geweest zijn, dus

$$2^{x-1} < n \Rightarrow x < (\log n) + 1$$

In verdubbelstap i worden 2^{i-1} elementen gekopieerd, samen zijn dat dus

$$\sum_{i=1}^{\lfloor \log n \rfloor + 1} 2^{i-1} = \sum_{i=0}^{\lfloor \log n \rfloor} 2^i < 2 \cdot 2^{\lfloor \log n \rfloor} \leq 2n$$

.
Dus zijn er samen ten hoogste $2n$ kopieerstappen en n verwijder/toevoegstappen nodig. Dat is een bovengrens van $3n$ voor de hele reeks van bewerkingen. Inderdaad hebben wij dus ook hier een veel betere grens bewezen dan in het geval waar wij voor elke bewerking het slechtste geval stellen. Natuurlijk zijn de voorbeelden ook op een manier gekozen dat het werkt!

De aggregaatmethode is vooral toepasbaar in gemakkelijke gevallen (zoals onze voorbeelden). In moeilijke gevallen zijn de volgende methoden vaak beter toepasbaar.

Oefening 24 *De bedoeling van deze oefening is om in te zien dat de technieken die wij geleerd hebben wel degelijk intuïtief zijn. Het gaat over een onderwerp dat jullie al kennen en waar jullie ook de complexiteit reeds van kennen. Door het toepassen van de nieuwe methoden op een bekend en gemakkelijk geval zien jullie hopelijk gemakkelijker wat het idee achter deze technieken is.*

Gegeven zijn twee enkelvoudig gelinkte gesorteerde lijsten met elk n objecten. Dus: je kent de twee eerste objecten en elk object bevat een referentie naar zijn opvolger. De tweede lijst moet gemerged worden in de eerste lijst zodat er op het einde een enkele gesorteerde lijst is. Dat gebeurt door een plaatsoperatie: eerst wordt de plaats in de eerste lijst gezocht, waar het eerste element van de tweede lijst geplaatst moet worden en dan wordt het daar geplaatst. De kost van deze operatie is het aantal vergelijkingen dat je moet doen. Als je de plaats voor het i-de element hebt gevonden, vertrek je van daar en zoek je de plaats voor het $(i+1)$-de element.

- *Toon aan dat het plaatsen van één element in het slechtste geval $\Theta(n)$ stappen kan vragen (wat tot een bovengrens van de complexiteit van $O(n^2)$ aanleiding zou geven als je de* slechtste geval maal aantal bewerkingen *methode zou toepassen).*

- *Is er een vast getal (dus een constante) k zodat er ten hoogste k plaatsoperaties kunnen zijn die tijd $\Theta(n)$ vragen als je twee lijsten merget?*

  indien ja: *bepaal k*

  anders: *toon aan dat k niet bestaat*

- *Toon d.m.v. de aggregaatmethode aan dat de geamortiseerde complexiteit van alle n invoegbewerkingen gemiddeld $O(1)$ per bewerking is.*

3.2 De accounting methode

De accounting methode kan je zo interpreteren alsof het algoritme geld kreeg voor elke bewerking. Wij noemen deze betaling de *toegekende kost*. Met dit geld worden de kosten van de bewerkingen betaald. Sommige kosten worden direct betaald en soms is er nog geld over dat je op een rekening kan bewaren. Soms kost een bewerking meer dan je betaald krijgt. Dan moet er voldoende geld op een rekening zijn om het verschil te betalen. De rekeningen worden meestal aan elementen van de datastructuur toegekend – maar ook dat is maar een hulpmiddel voor de argumentatie om er altijd een zicht op te hebben welke rekeningen er zijn.

Het principe is dus als volgt:

- Wij werken met twee soorten kost: de echte kost en de toegekende kost (de betaling). De echte kost wordt door het probleem bepaald maar de toegekende kost is een **theoretische kost** die je zelf moet vastleggen. Het is gewoon een **hulpmiddel**.
- Alle bewerkingen worden nu door de toegekende kost en/of de inhoud van de rekeningen betaald. Als je maar een deel van de toegekende kost gebruikt om een bewerking te betalen, kan je de rest dus aan rekeningen toekennen (het wordt op een spaarrekening opgeslagen).
- Als de toegekende kost lager is dan de echte kost, kan de bewerking alleen maar doorgaan als het verschil uit een rekening betaald kan worden. Meestal worden in de argumentatie de rekeningen van objecten die bij de bewerking betrokken zijn, gebruikt.

Als wij nu kunnen aantonen dat alle bewerkingen gedaan kunnen worden dan is de som van de echte kosten dus op elk moment ten hoogste even groot als de som van de toegekende kosten. Tenslotte werden de bewerkingen daarmee betaald en anders zou er een bewerking zijn waar de echte kost hoger is dan de toegekende kost en die je niet van de opgeslagen kredieten kan betalen. De som van de toegekende kosten is dus een bovengrens voor de geamortiseerde complexiteit.

Vraag: geldt dat alleen maar voor de **som** of is de aan een enkele bewerking toegekende kost ook een bovengrens voor de echte kost van de bewerking?

Dat kan je ook formeel schrijven. Stel dat k_i de echte kost van bewerking i is en t_i de toegekende kost. Bovendien zij r_j het krediet op rekening j en $1, \ldots, k$ de bestaande rekeningen na stap m. Dan is de echte kost na m stappen $\sum_{i=1}^{m} k_i$. De toegekende kosten komen allemaal op een rekening terecht of worden door een bewerking opgebruikt, dus

$$\sum_{i=1}^{m} t_i \geq \sum_{i=1}^{m} k_i + \sum_{i=1}^{k} r_i$$

Hier heb je al $\geq$ omdat in de methode soms rekeningen met positieve kredieten verdwijnen – bv. als de rekeningen als aan elementen toebehorend geïnterpreteerd worden en die elementen verwijderd werden. Omdat de r_i allemaal groter dan of gelijk aan 0 zijn, geldt dan

$$\sum_{i=1}^{m} t_i \geq \sum_{i=1}^{m} k_i$$

Hier zie je ook hoe je *in principe* negatieve kredieten zou kunnen toelaten en toch een bovengrens krijgen, maar wij zullen alleen maar het geval bespreken van kredieten die altijd positief zijn.

Toegepast op voorbeeld 1 – de binaire teller:

	echte kost	toegekende kost
1 bijtellen	aantal k gewijzigde bits	2 (uitleg volgt)

De echte kost is gegeven en maakt deel uit van het probleem. De toegekende kost hebben wij vastgelegd en nu moeten wij aantonen dat met deze toegekende kost inderdaad alle bewerkingen uitgevoerd kunnen worden.
Onze argumentatie is als volgt: In elke bewerking wordt ten hoogste één variabele van 0 tot 1 gewijzigd. Inderdaad is het behalve in het geval van een overflow altijd precies 1 variabele. De verdeling van onze kosten interpreteren wij als volgt: Voor de wijziging van een variabele met waarde 0 gebruiken wij 1 van onze toegekende kost en de overblijvende 1 van de 2 eenheden slaan wij als krediet op bij de 1 die wij net gewijzigd hebben.
Zo kan je zien dat alle bewerkingen betaald kunnen worden: Wij gebruiken de toegekende kost alleen voor het wijzigen van nullen – dus kunnen wij ervoor zorgen dat elke variabele met waarde 1 inderdaad krediet 1 heeft. Het wijzigen van de variabelen met waarde 1 kunnen de variabelen dan uit hun kredieten betalen: elke 1 heeft krediet 1 om voor zichzelf te betalen.
Wij hebben dus opnieuw bewezen dat de geamortiseerde complexiteit ten hoogste $2 \cdot 2^n$ is – dus gemiddeld een constante kost per bewerking.
Hier hebben wij gebruikt dat de teller in het begin 0 is. Als dat niet zo zou zijn dan zouden er enen inzitten die geen krediet hebben en als die gewijzigd moeten worden zou je de bewerking dus niet kunnen betalen!
Wat wij niet nodig hadden, was dat er precies 2^n bewerkingen zijn. Inderdaad is dat ook voor de aggregaatmethode niet nodig – het maakte alleen de argumentatie iets duidelijker.

Oefening 25 *Stel dat een binaire teller in het begin* 0 *is. Bewijs dat de geamortiseerde kost van een bewerking in een reeks van n operaties op deze*

binaire teller $O(1)$ is. Pas de aggregaatmethode toe. Stel deze keer niet dat n een macht van 2 is.

Toegepast op voorbeeld 2 – een array groter maken:

	echte kost	toegekende kost
een element toevoegen	$1 + k$ waarbij k het aantal gekopieerde elementen is	3 (uitleg volgt)
een element verwijderen	1	1 (uitleg volgt)

Ook hier moeten wij nu aantonen dat de bewerkingen altijd betaald kunnen worden. In het geval van een verwijderoperatie is dat gemakkelijk: de echte kost is 1 en de toegekende kost ook. Wij slaan dus geen krediet op maar gebruiken ook geen krediet.
In het geval van een toevoegbewerking gebruiken wij 1 voor het toevoegen van het element en slaan 2 als krediet bij het element op. Het kopiëren wordt nu altijd volledig uit kredieten betaald: In het geval dat maar 1 element gekopieerd moet worden, heeft het duidelijk voldoende krediet. Als $k \geq 2$ elementen gekopieerd moeten worden, dan werd sinds het laatste kopiëren (waar voor sommige elementen de kredieten werden opgebruikt) ten minste de tweede helft van de array nieuw toegevoegd (misschien werden ook andere elementen verwijderd en opnieuw toegevoegd). De elementen in de tweede helft hebben dus allemaal krediet 2 en kunnen het kopiëren voor zich (daarvoor gebuiken ze 1) en voor de elementen in de eerste helft (daarvoor wordt ook 1 van het krediet gebruikt) betalen. Omdat in de eerste en tweede helft even veel elementen zitten, kunnen dus alle kopieerbewerkingen betaald worden.
Alle bewerkingen kunnen dus betaald worden en de geamortiseerde kost voor m bewerkingen op een initieel lege array is dus ten hoogste $3 * m$ omdat 3 een bovengrens is voor de toegekende kost per bewerking.

Waar gebruiken wij hier dat wij van een lege array vertrekken?

3.3 De potentiaalmethode

Voor de potentiaalmethode definiëren wij een potentiaal van een datastructuur. Hoewel je de potentiaal misschien kan interpreteren als een spaarrekening van de hele datastructuur zijn er twee belangrijke verschillen met de accounting methode:

(i) De potentiaal hangt alleen van de datastructuur af. Je kent geen potentiaal of krediet of wat dan ook toe aan de bewerkingen! Maar een bewerking heeft wel invloed op de datastructuur en dus op de potentiaal. Als de datastructuur door de bewerking gewijzigd wordt, verandert dus in de meeste gevallen ook de potentiaal. Hoe groot de verandering is, moet je berekenen.

(ii) Er is maar één potentiaal voor de hele datastructuur en niet zo als in de accounting methode vele kleine spaarrekeningen.

De potentiaal lijkt dus op de potentiaal uit de fysica: als je met een fiets op een heuvel staat, dan is de potentiële energie die je op dat punt hebt alleen afhankelijk van het punt (precies: de hoogte van het punt) en niet de manier waarop je naar dat punt bent gekomen.

Oefening 26 *Geef een voorbeeld van 2 reeksen van bewerkingen voor de array die uitgebreid wordt, waarbij het resultaat dezelfde array is met dezelfde inhoud maar waar de inhoud van de rekeningen (zoals gebruikt in ons bewijs in het deel van de accounting methode) van de elementen (en ook de som van de inhouden) verschillend zijn. De potentiaal moet natuurlijk in beide gevallen dezelfde zijn omdat de array en de inhoud identiek zijn.*

Het principe van de potentiaalmethode kan je als volgt beschrijven:

- Je definieert een potentiaal als functie van de datastructuur. Als de datastructuur D is, schrijven wij normaal $\Phi(D)$. Deze potentiaal mag alleen van de structuur en de inhoud van de datastructuur afhangen. Een goede potentiaal vinden is normaal het moeilijkste gedeelte als je de potentiaalmethode wil toepassen. Daarna is het "alleen" nog rekenen...

- Als een bewerking met kost k op een datastructuur D gebeurt en die verandert daardoor in een datastructuur D' (die ook gelijk kan zijn), dan definiëren wij de gewijzigde kost g als $g = k + \Phi(D') - \Phi(D)$. Deze gewijzigde kost is ook een **virtuele** kost en **geen echte** kost.

- De som van de gewijzigde kosten is in de meeste gevallen een bovengrens voor de som van echte kosten. Soms is er nog een correctieterm (uitleg volgt). Maar de gewijzigde kost van een enkele bewerking is geen bovengrens voor de echte kost van een enkele bewerking!

Je kan de gewijzigde kost ook zo interpreteren dat er soms iets *door de potentiaal betaald wordt* (als de potentiaal daalt) en je soms iets *in de potentiaal opslaat* (als de potentiaal stijgt). Het lijkt dus (in dit opzicht) op de accounting methode.
Wij kijken nu naar een reeks van m bewerkingen waarbij bewerking i een kost van k_i en een gewijzigde kost van g_i heeft. Wij noemen de initiële datastructuur D_0 en de datastructuur na de i-de bewerking D_i. Dan geldt:

$$\begin{aligned}\sum_{i=1}^{m} g_i &= \sum_{i=1}^{m}(k_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{m} k_i + \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^{m-1}(-\Phi(D_i) + \Phi(D_i)) \\ &= \sum_{i=1}^{m} k_i + (\Phi(D_m) - \Phi(D_0))\end{aligned}$$

Dus is $\sum_{i=1}^{m} g_i - (\Phi(D_m) - \Phi(D_0))$ een bovengrens voor de geamortiseerde kost van de reeks. In het geval dat $\Phi(D_m) \geq \Phi(D_0)$ is de som van de gewijzigde kosten alleen al een bovengrens voor de geamortiseerde kost van de reeks.

Toegepast op voorbeeld 1 – de binaire teller:

De belangrijkste stap is het vastleggen van een potentiaal. Voor een gegeven toestand D van de datastructuur – dat betekent gegeven waarden van de variabelen $x_{n-1}, \ldots, x_0$ – definiëren wij de potentiaal $\Phi(D)$ als het aantal variabelen met waarde 1.
Nu moeten wij berekenen wat de gewijzigde kosten zijn:

	echte kost	gewijzigde kost
1 bijtellen	aantal k gewijzigde bits	$k + \Phi(D') - \Phi(D)$ ≤ 2 (uitleg volgt)

Hier staat opnieuw D voor de datastructuur vóór de bewerking en D' voor de datastructuur na de bewerking. Als k bits gewijzigd worden dan weten wij precies wat er gebeurt: Ten hoogste 1 bit wordt van 0 naar 1 gewijzigd en ten minste $k-1$ bits van 1 naar 0. Dus geldt $\Phi(D') - \Phi(D) \leq 1 - (k-1)$ en dus $k + \Phi(D') - \Phi(D) \leq 2$.

In het begin is de potentiaal 0 en op het einde 0 als er precies 2^n bewerkingen zijn en anders positief. Dus is de som van de gewijzigde kosten een bovengrens voor de geamortiseerde complexiteit van de reeks bewerkingen. De som van de gewijzigde kosten is voor m bewerkingen dus $2 \cdot m$. Omdat deze som een bovengrens is voor de geamortiseerde complexiteit hebben we bewezen dat de geamortiseerde complexiteit ten hoogste $2 \cdot m$ is – dus gemiddeld een constante kost per bewerking.

Toegepast op voorbeeld 2 – een array groter maken:

Hier definiëren wij voor een gegeven array D met lengte l de potentiaal als

$$\Phi(D) = -l$$

Een informele redenering waarom dat een goede keuze is, is als volgt: je wil de gewijzigde kost laag houden omdat dat (samen met de correctieterm) jouw bovengrens voor de echte kost vormt en een lagere bovengrens is een betere bovengrens. Dat betekent dat in gevallen waar de echte kost k hoog is, de potentiaal moet dalen zodat $\Phi(D') - \Phi(D)$ ervoor kan zorgen dat de gewijzigde kost $k + \Phi(D') - \Phi(D)$ klein blijft. In gevallen waar k klein is, mag de potentiaal zelfs stijgen. Als je dus een langere array kiest, moet de potentiaal dalen. Zo zie je al ongeveer hoe de potentiaal er moet uitzien. Maar wij moeten natuurlijk aantonen dat het met deze definitie ook werkt:

	echte kost	gewijzigde kost
een element toevoegen	$1 + k$ waarbij k het aantal gekopieerde elementen is	$1 + k + \Phi(D') - \Phi(D)$ $= 1 + k - 2k + k = 1$ (uitleg volgt)
een element verwijderen	1	$1 + \Phi(D') - \Phi(D)$ $= 1$ (uitleg volgt)

Als tijdens het toevoegen geen elementen gekopieerd moeten worden (dus $k = 0$), dan is het duidelijk dat l niet verandert. Maar als er elementen gekopieerd moeten worden dan groeit ook altijd de lengte van de array. Als er k elementen gekopieerd moeten worden, was de oude lengte k en de nieuwe lengte is $2 \cdot k$ – dus $\Phi(D) = -k$ en $\Phi(D') = -2k$.
In het begin is de potentiaal -1. Maar op het einde kan de potentiaal duidelijk kleiner zijn dan -1! Stel dat de lengte na de m stappen $l_m > 1$ is.

Dan werd de array uitgebreid omdat er meer dan $l_m/2$ elementen in geplaatst werden, dus is het aantal bewerkingen m groter dan $l_m/2$ en dus $l_m < 2m$.

$$\begin{aligned}\textstyle\sum_{i=1}^{m} g_i - (\Phi(D_m) - \Phi(D_0)) &\leq m - ((-l_m) - (-1)) \\ &= m + l_m - 1 < m + 2m = 3m\end{aligned}$$

Dus hebben wij opnieuw een geamortiseerde complexiteit die gemiddeld per bewerking constante kost heeft.
Wij wilden hier vooral tonen wat je kan doen als de potentiaal op het einde niet groter is dan in het begin, maar Oefening 28 toont dat het ook anders kan.

Oefening 27 *Definieer de potentiaal voor voorbeeld 2 als*

$$\Phi(D) = 2e - l$$

voor een gegeven array D met lengte l dat e elementen bevat.
Bepaal een grens voor de geamortiseerde complexiteit aan de hand van deze potentiaal.

Oefening 28 *Kan je de potentiaal voor voorbeeld 2 ook zo kiezen dat de potentiaal op het einde altijd groter of gelijk is aan de potentiaal in het begin en je toch al een constante gemiddelde kost per bewerking krijgt?*

Oefening 29 *Gegeven de binaire teller zoals in Voorbeeld 1 – alleen dat de mogelijke bewerkingen niet alleen* één bijtellen *is, maar* één , twee of drie bijtellen*. Geef een bovengrens voor de geamortiseerde complexiteit van een reeks van n bewerkingen op een binaire teller die initieel* 0 *is. Geef niet alleen de $O()$ notatie maar – net zoals in de voorbeelden – ook een constante. De constante moet beter zijn dan de* 6 *die je krijgt door bv.* drie bijtellen *gewoon te schatten door drie keer één bij te tellen.*
Je mag om het even welke methode gebruiken om de grens te bewijzen.

Oefening 30 *Zie Oefening 24.*

Gegeven zijn twee enkelvoudige gelinkte gesorteerde lijsten met elk n objecten. Dus: je kent de twee eerste objecten en elk object bevat een referentie naar zijn opvolger. De tweede lijst moet gemerged worden in de eerste lijst zodat er op het einde een enkele gesorteerde lijst is. Dat gebeurt door een plaats-operatie: eerst word de plaats in de eerste lijst gezocht, waar het eerste element van de tweede lijst geplaatst moet worden en dan wordt het daar geplaatst. De kost van deze operatie is het aantal vergelijkingen dat je moet doen. Als je de plaats voor het i-de element hebt gevonden, vertrek je van daar en zoek je de plaats voor het $(i+1)$-de element.

- *Toon d.m.v. de potentiaalmethode aan dat de geamortiseerde complexiteit van alle n invoegbewerkingen samen $O(n)$ is.*

Oefening 31 *Een stapel met een multipop bewerking. Je hebt de volgende bewerkingen;*

S.push(x): *voegt sleutel x toe aan de stapel S. De kost van deze bewerking is constant, dus $\Theta(1)$.*

S.pop(): *verwijdert de bovenste sleutel van S. De kost van deze bewerking is ook constant, dus $\Theta(1)$.*

S.multipop(k): *verwijdert de bovenste k sleutels van S als S ten minste k sleutels bevat en anders alle sleutels in S. De kost van deze bewerking is proportioneel met het aantal verwijderde sleutels, dus $\Theta(k')$ waarbij $k' = \min\{k, |S|\}$ met $|S|$ het aantal sleutels in S, waarbij wij stellen dat $|S| > 0$.*

Bereken de geamortiseerde complexiteit voor een reeks van n bewerkingen op een initieel lege stapel op elke van de 3 geziene manieren.

Oefening 32 *Als je elke keer als je een array groter maakt 2 keer zoveel geheugen alloceert, groeit het geheugen dus exponentieel. Om dat te vermijden kiezen wij er nu voor om het geheugen langzamer uit te breiden. Wij onderzoeken twee manieren van doen:*

- *Je begint met een array van grootte 100 en kiest als de array te klein wordt de nieuwe array altijd 20% groter dan de oude (waarbij je naar boven afrondt als het geen geheel getal is). Op deze manier groeit de array nog altijd exponentieel maar natuurlijk veel trager.*

- *Je begint met een array van grootte 100 en kiest als de array te klein wordt de nieuwe array altijd 100 eenheden groter dan de oude. Op deze manier groeit het geheugen niet exponentieel snel.*

Bereken voor beide manieren van doen de geamortiseerde tijd voor een reeks van m bewerkingen op een initieel lege array. Kies zelf welke techniek je wil toepassen. Zijn de grenzen die je vindt goede grenzen?

Oefening 33 *Wat is de geamortiseerde complexiteit van een reeks van n bewerkingen op een initieel lege binaire hoop? Bewijs dat jouw antwoord een goede bovengrens is.*

Oefening 34 *Stel dat jouw binaire teller in het begin niet* 0 *is, maar de bits* `1 0 1 0 1...` `1 0 1 0` *bevat en je hem incrementeert tot je een overflow hebt en alles* 0 *is. Is de gemiddelde kost per bewerking nog altijd constant? (Natuurlijk moet je aantonen dat jouw antwoord juist is – dat wordt vanaf nu niet altijd herhaald.)*

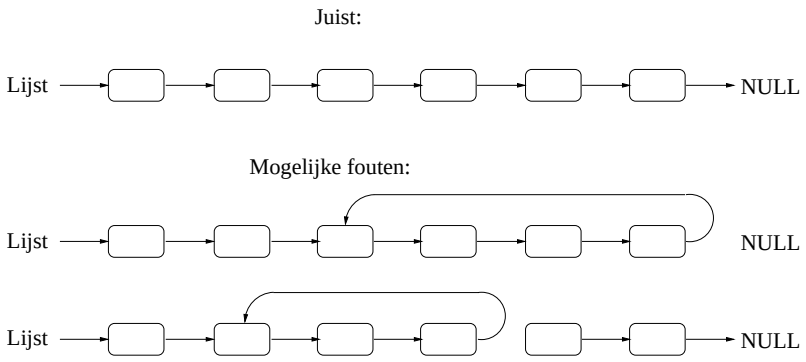
Oefening 35 *Wij hebben al oefeningen opgelost, waar je een array groter maakt indien dat nodig is. Maar als je echt geheugen wil besparen moet je het ook kleiner maken als je ziet dat je zo veel geheugen niet nodig hebt. Dus stel dat je als volgt werkt: Je begint met een array van 1000 elementen. Toevoegen doe je altijd op het einde van de array en je verwijdert altijd het laatste element. Elke keer dat je meer elementen wil toevoegen dan in de array passen, alloceer je dubbel zo veel ruimte voor de array en kopieer je de oude array. Als de grootte n was, vraagt dat n stappen om de array te kopiëren en 1 stap om het nieuwe element te plaatsen. Maar als je ziet dat je minder dan de helft van de array gebruikt (en de lengte is groter dan 1000) alloceer je een kleinere array met grootte drie kwart van de originele array (dus de helft van de vrije plaatsen) en kopieer je de n elementen naar de nieuwe array. Dat kost n stappen.*
Toon aan dat een reeks van m bewerkingen op een initieel lege array gemiddeld constante tijd vraagt. Gebruik ofwel de accounting-methode ofwel de potentiaalmethode.

Tips:

accounting-methode: *Denk er eerst over na, wat volgens jou de duurste reeks van bewerkingen zou zijn. Definieer dan de toegekende kost zo dat er in dit geval voldoende krediet is om de dure bewerkingen te betalen. Zo krijg je een goed idee wat de toegekende kost moet zijn. Maar: je moet aantonen dat er* **altijd** *voldoende krediet is – en niet alleen als er een zekere reeks van bewerkingen is.*

potentiaalmethode *: Denk er eerst over na in welke gevallen dure bewerkingen kunnen gebeuren. Dat zijn gevallen waar de potentiaal groot moet zijn. Is er een geval waar twee dure bewerkingen (bijna) direct na elkaar kunnen gebeuren? Indien ja moet de potentiaal dan het grootst zijn. In welke situatie ben je min of meer veilig – dat betekent dat er tenminste voor meerdere stappen geen dure bewerkingen kunnen gebeuren? In deze situatie kan de potentiaal 0 zijn. Parameters die* **misschien** *nuttig zijn om de potentiaal te definiëren zijn de lengte n van de array, het aantal vrije posities met index tussen* $(n/2) + 1$ *en* $(2/3)n$ *en het aantal vrije posities met index tussen* $(2/3)n + 1$ *en* n*.*

Oefening 36 *Je werkt met een enkelvoudig gelinkte lijst – dus een lijst waar je een pointer naar het beginelement hebt en elk element heeft een pointer naar het volgende element. Het laatste element heeft een pointer naar NULL. Maar nu werd vastgesteld dat één van de programma's die jouw lijst gebruikt een fout in de lijst veroorzaakt: soms worden pointers gewijzigd en wijzen in plaats van naar NULL of de volgende buur naar een vroeger element in de lijst (zie Figuur 16). Jouw taak is nu een efficiënt programma te schrijven dat kan detecteren of de lijst nog in orde is. Efficiënt betekent hier: als n het aantal elementen in de lijst is dan moet jouw programma in tijd $O(n)$ draaien. Bovendien mag het maar $O(1)$ extra geheugen vragen en mag het de lijst niet wijzigen. Ontwerp een dergelijk programma en bewijs dat het aan de eisen voldoet.*



Figuur 16:

3.4 De geamortiseerde complexiteit van semi-splay

In dit hoofdstuk bewijzen wij dat een reeks van n bewerkingen op een initieel lege semi-splay boom een geamortiseerde complexiteit van $O(\log n)$ heeft. Wij zullen herhaaldelijk gebruiken dat log() monotoon stijgend is. Als $x \leq y$ dan is dus $\log x \leq \log y$.
Wij hebben de volgende ongelijkheid nodig:

Opmerking 9 *Stel dat a, b, c natuurlijke getallen zijn en dat ze allemaal verschillen van 0. Als dan $a + b \leq c$ geldt dan is* $\log a + \log b \leq 2 \log c - 2$.

Oefening 37 *Bewijs Opmerking 9. Omdat als $a + b \leq c$ zeker geldt dat $b \leq c - a$ en dus $\log a + \log b \leq \log a + \log(c - a)$ is het voldoende het maximum van de functie $\log x + \log(c - x)$ voor een constante c te berekenen op het interval $[1, c - 1]$. Hiervoor kan je je kennis uit het middelbaar onderwijs toepassen!*

Stelling 10 *Een reeks van $n > 1$ bewerkingen op een initieel lege semi-splay boom heeft een geamortiseerde complexiteit van $O(n \log n)$, dus is de gemiddelde kost $O(\log n)$ per bewerking.*
Hierbij telt de kost *het aantal bezochte toppen.*

Er is een voor de hand liggende manier om de bewerkingen op te splitsen in elementaire bewerkingen (deelboom vervangen, een top toevoegen, een top verwijderen), die allemaal constante kost hebben. Wij zouden dus kunnen proberen het aantal elementaire stappen te tellen en de aggregaatmethode toepassen – maar het is niet duidelijk hoe je deze stappen kan tellen. Hier is de krachtigere potentiaalmethode nuttig!

Bewijs: Wij gebruiken dus de potentiaalmethode. Voor een lege boom definiëren wij $\Phi(T) = 0$. Stel nu dat T niet leeg is. Voor een top v in een semi-splay boom T definiëren wij $A_T(v)$ ($A()$ voor aantal) als het aantal toppen in de deelboom bestaande uit v en al zijn nakomelingen in T. Bovendien zij $L_T(v) = \log A_T(v)$ (met $L()$ voor logaritme). De potentiaal van de hele semi-splay boom T definiëren wij nu als

$$\Phi(T) = \sum_{v \in T} L_T(v)$$

De potentiaal wordt gewijzigd als een top toegevoegd wordt (nog zonder de boom te herbalanceren), verwijderd wordt (zonder herbalanceren) of als tijdens het herbalanceren deelbomen vervangen worden. Wij zullen deze 3 delen onafhankelijk van elkaar onderzoeken. Eerst het gemakkelijkste:

Deelresultaat 1: Als een blad of een top met maar 1 kind uit een boom T wordt verwijderd en het resultaat is de boom T' dan geldt $\Phi(T') \leq \Phi(T)$.

Hierbij bedoelen wij alleen maar het verwijderen van de top – zonder de splaybewerkingen die achteraf ook nog moeten gebeuren! Maar dan is het duidelijk omdat de aantallen van toppen in de deelbomen alleen maar kleiner kunnen worden.

Deelresultaat 2: Als een nieuwe top tot de boom T wordt toegevoegd en het resultaat is de boom T' dan geldt $\Phi(T') - \Phi(T) \leq \log(|T'|)$.

Ook hier hebben wij het alleen over het toevoegen zelf – zonder de splaybewerkingen die achteraf ook nog moeten gebeuren!

Als T leeg was, klopt het duidelijk, dus stel dat T niet leeg was.

Stel dat de toppen op het pad van de wortel naar de nieuwe top $v_1, \ldots, v_k$ zijn, waarbij v_k de nieuwe top is. Natuurlijk zijn alleen maar de bijdragen van de toppen op dit pad gewijzigd – voor alle andere toppen is het aantal opvolgers gelijk gebleven. Dan geldt

$$\begin{aligned}\Phi(T') &= \Phi(T) - \textstyle\sum_{i=1}^{k-1} L_T(v_i) + \sum_{i=1}^{k} L_{T'}(v_i) \\ &= \Phi(T) + L_{T'}(v_1) + \textstyle\sum_{i=1}^{k-1}(L_{T'}(v_{i+1}) - L_T(v_i))\end{aligned}$$

Maar voor $i \geq 1$ geldt $A_T(v_i) \geq A_T(v_{i+1}) + 1 = A_{T'}(v_{i+1})$. Dat geeft $L_T(v_i) \geq L_{T'}(v_{i+1})$ en dus $L_{T'}(v_{i+1}) - L_T(v_i) \leq 0$ en ten slotte

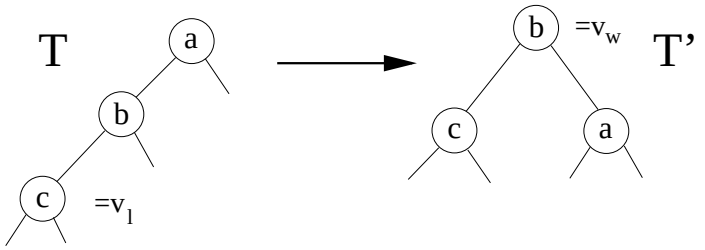
$$\Phi(T') \leq \Phi(T) + L_{T'}(v_1) = \Phi(T) + \log(|T'|)$$

omdat de wortel natuurlijk precies $|T'|$ opvolgers (de wortel zelf meegeteld) heeft.

Deelresultaat 3: Stel dat een deelboom in een boom T op een semi-splay manier wordt vervangen en het resultaat is T'. Wij noemen de laagste top in de deelboom van T die wordt vervangen v_l (l voor *laag*) en de top die de nieuwe wortel van de nieuwe deelboom van T' wordt v_w (w voor *wortel*). Natuurlijk kan dat dezelfde top zijn!

Dan geldt:

$\Phi(T') - \Phi(T) \leq 2L_{T'}(v_w) - 2L_T(v_l) - 2.$



Figuur 17: Eén van de mogelijkheden hoe een deelboom wordt vervangen.

Wij bespreken eerst het geval dat in Figuur 17 voorgesteld wordt. In dat geval is $c = v_l$ en $b = v_w$. De enige toppen waarvoor $L()$ misschien kan veranderen, zijn a, b en c. Wij hebben dus

$$\Phi(T') = \Phi(T) + (L_{T'}(a) + L_{T'}(b) + L_{T'}(c)) - (L_T(a) + L_T(b) + L_T(c))$$

Om de aantallen $A()$ te schatten moeten wij naar Figuur 17 kijken. Wij zien onmiddellijk:

$A_{T'}(b) = A_T(a) \Rightarrow L_{T'}(b) = L_T(a)$

en dus

$$\Phi(T') = \Phi(T) + (L_{T'}(a) + L_{T'}(c)) - (L_T(b) + L_T(c))$$

Bovendien

$A_T(c) < A_T(b) \Rightarrow L_T(c) < L_T(b)$

en dus

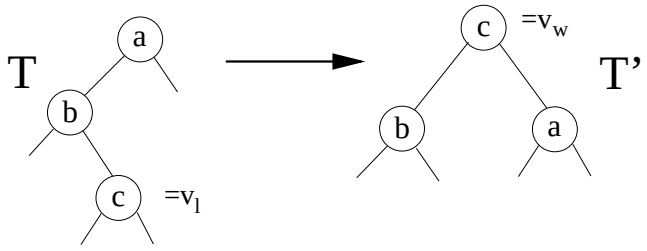
$$\Phi(T') \leq \Phi(T) + (L_{T'}(a) + L_{T'}(c)) - 2L_T(c)$$

Wat wij ook in de figuur zien, is dat $A_{T'}(a) + A_{T'}(c) + 1 = A_{T'}(b)$, dus $A_{T'}(a) + A_{T'}(c) \leq A_{T'}(b)$ en met Opmerking 9 volgt dan $\log A_{T'}(a) + \log A_{T'}(c) \leq 2 \log A_{T'}(b) - 2$ dus $L_{T'}(a) + L_{T'}(c) \leq 2L_{T'}(b) - 2$. Daarmee hebben wij

$$\Phi(T') \leq \Phi(T) + 2L_{T'}(b) - 2L_T(c) - 2$$

en dat is precies wat wij moesten bewijzen.

De andere gevallen zijn volledig analoog. Omdat het voor het duidelijk gelijkaardig geval dat het pad twee keer rechts afslaat zeker nog gemakkelijker is om te zien dan een zigzag-geval werken wij dit nog expliciet uit:



Figuur 18: De tweede van de mogelijkheden hoe een deelboom wordt vervangen.

In het in Figuur 18 voorgestelde geval is $v_l = v_w = c$

Wij kunnen aflezen

$A_{T'}(c) = A_T(a) \Rightarrow L_{T'}(c) = L_T(a)$

$A_T(c) < A_T(b) \Rightarrow L_T(c) < L_T(b)$

en

$A_{T'}(a) + A_{T'}(b) + 1 = A_{T'}(c) \Rightarrow L_{T'}(a) + L_{T'}(b) \leq 2L_{T'}(c) - 2$

en deze 3 observaties geven

$$\begin{aligned}\Phi(T') &= \Phi(T) + (L_{T'}(a) + L_{T'}(b) + L_{T'}(c)) - (L_T(a) + L_T(b) + L_T(c)) \\ &\leq \Phi(T) + 2L_{T'}(c) - 2L_T(c) - 2\end{aligned}$$

en dat is precies wat wij wilden aantonen.

Natuurlijk kan je de argumenten ook algemeen – zonder gevallen – toepassen door rekening te houden met de *hoogte* van de toppen. Maar aan de hand van de figuren in de speciale gevallen is het misschien iets gemakkelijker te verstaan wat er gebeurt...

Deelresultaat 4: Als tijdens een toevoegbewerking, een opzoekbewerking of een verwijderbewerking in een boom T langs een pad met $2s+1$ toppen deelbomen vervangen worden en het resultaat is de boom T' dan geldt

$$\Phi(T') - \Phi(T) \leq 2\log|T'| - 2s$$

Hierbij worden alleen de toppen geteld die echt betrokken zijn bij het vervangen van de deelbomen. Als de wortel van de boom niet betrokken is, wordt die ook niet meegeteld.

Wij vervangen s keer een deelboom. Stel dat de boom na de i-de vervanging de boom T_i is en dus $T_0 = T$ en $T_s = T'$. Bovendien zij $v_{l,i}$ de laagste top in de deelboom die in stap i wordt vervangen en $v_{w,i}$ de wortel van de nieuwe deelboom die in stap i wordt geplaatst.

Dan is

$$\begin{aligned}\Phi(T') &= \Phi(T) + \textstyle\sum_{i=1}^{s}(\Phi(T_i) - \Phi(T_{i-1})) \\ &\leq \Phi(T) + \textstyle\sum_{i=1}^{s}(2L_{T_i}(v_{w,i}) - 2L_{T_{i-1}}(v_{l,i}) - 2)\end{aligned}$$

Maar de top die de nieuwe wortel van een net geplaatste deelboom is, is de laagste top in de volgende stap, dus $v_{w,i} = v_{l,i+1}$. Als wij dat invullen, krijgen wij

$$\begin{aligned}\Phi(T') &\leq \Phi(T) + 2L_{T_s}(v_{w,s}) + \textstyle\sum_{i=1}^{s-1}(2L_{T_i}(v_{w,i}) - 2L_{T_i}(v_{l,i+1})) - 2L_{T_0}(v_{l,1}) - 2s \\ &\leq \Phi(T) + 2L_{T_s}(v_{w,s}) - 2s \leq \Phi(T) + 2\log|T'| - 2s\end{aligned}$$

Nu hebben wij voor alle deelbewerkingen die de potentiaal wijzigen bovengrenzen voor het verschil van de potentiaal vóór de bewerking en erna. Als een top wordt toegevoegd wordt de potentiaal door het toevoegen en het splayen gewijzigd, als een top wordt verwijderd wordt de potentiaal door het verwijderen en het splayen gewijzigd en bij het opzoeken enkel door het splayen. Nu kunnen wij onze tabel maken met T' de boom na de bewerking en T de boom vóór de bewerking. De toevoeg- en verwijder-bewerking splitsen wij nog verder op en schrijven $\bar{T}$ voor de boom na het toevoegen of verwijderen maar nog voordat gesplayed wordt. Als het aantal toppen van het pad dat wordt doorlopen k is, dan worden ofwel langs een pad met $2s+1 = k$ toppen ofwel langs een pad met $2s+1 = k-1$ toppen deelbomen vervangen waarbij $2s$ zoals in deelresultaat 4 is. In elk geval geldt $k \leq 2s+2$.

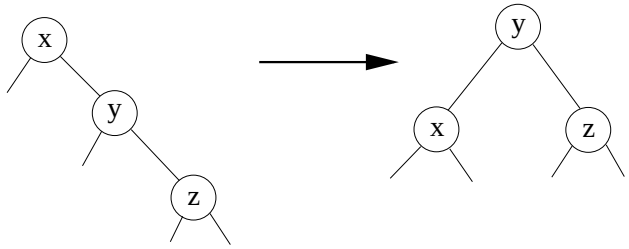
	echte kost	gewijzigde kost
top opzoeken	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ nu wordt deelresultaat 4 gebruikt: $\leq k + 2\log\|T'\| - 2s$ $\leq 2\log\|T'\| + 2$
top verwijderen	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ $= k + \Phi(T') - \Phi(\bar{T}) + \Phi(\bar{T}) - \Phi(T)$ nu wordt deelresultaat 1 gebruikt: $\leq k + \Phi(T') - \Phi(\bar{T})$ nu wordt deelresultaat 4 gebruikt: $\leq k + 2\log\|T'\| - 2s$ $\leq 2\log\|T'\| + 2$
top toevoegen	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ $= k + \Phi(T') - \Phi(\bar{T}) + \Phi(\bar{T}) - \Phi(T)$ nu wordt deelresultaat 2 gebruikt: $\leq k + \log\|\bar{T}\| + \Phi(T') - \Phi(\bar{T})$ $\|\bar{T}\| = \|T'\|$ en deelresultaat 4 geven: $\leq k + \log\|T'\| + 2\log\|T'\| - 2s$ $\leq 3\log\|T'\| + 2$

Maar omdat als je met een initieel lege boom begint na n stappen altijd

$|T| \leq n$ geldt, volgt nu onmiddellijk de stelling omdat de gewijzigde kost per bewerking $O(\log n)$ is en de potentiaal op het einde tenminste even groot is als in het begin.

■

Oefening 38 *Bewijs deelresultaat 3 expliciet voor het geval dat een deelboom vervangen wordt op de manier die je in Figuur 19 ziet.*

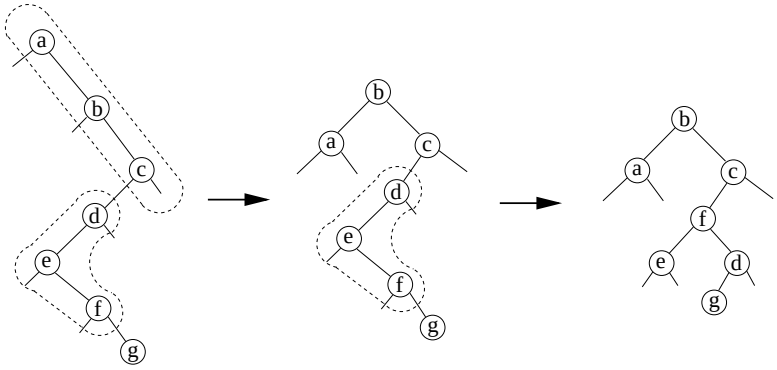


Figuur 19:

Het is natuurlijk analoog aan het eerste geval dat wij gezien hebben, maar het zelfstandig uit te werken helpt toch om het bewijs beter te verstaan.

Oefening 39 *Een andere variante van semi-splay is onafhankelijk semi-splay. Hier kies je de deelbomen die vervangen worden op een manier zodat ze geen toppen gemeen hebben. Dat kan je gemakkelijk topdown implementeren: stel dat $v_1, v_2, \ldots, v_n$ het pad is dat bv. tijdens het opzoeken (analoog toevoegen en verwijderen) wordt doorlopen. Als je een deelpad met 3 toppen hebt doorlopen (het eerste dergelijke deelpad is v_1, v_2, v_3), vervang je die door een complete binaire boom met 3 toppen. Het volgende deelpad begint dan met de* **volgende** *top (het tweede deelpad begint dus met v_4) en bevat geen top uit het eerste deelpad (daarom heet het* onafhankelijk*). Als er maar één of twee toppen over zijn wordt dit deelpad niet gewijzigd.*
Een voorbeeld wordt gegeven in Figuur 20
Bewijs of geef een tegenvoorbeeld voor de stelling:
Stelling: *Een reeks van $n > 1$ bewerkingen op een initieel lege onafhankelijk-semi-splay boom heeft een geamortiseerde complexiteit van $O(n \log n)$, dus is de gemiddelde kost $O(\log n)$ per bewerking.*

Oefening 40 *Je wilt voor jouw semi-splay boom behalve de toevoeg-, verwijder- en opzoekbewerking ook nog een iterator implementeren. Een dergelijke iterator heeft de bewerkingen* next() *en* reset()*. Iteratorbewerkingen hebben altijd de vorm van een reeks van next-bewerkingen afgesloten met een reset. Tussendoor zijn er geen toevoeg-, verwijder- of opzoekbewerkingen. De eerste*



Figuur 20:

next-bewerking geeft het kleinste element, de volgende het op één na kleinste, etc. Na een reset-bewerking kunnen terug andere bewerkingen gebeuren of een nieuwe reeks van iteratorbewerkingen. Een reset kan op elk moment gebeuren, maar als de next-bewerking het laatste (grootste) element teruggeeft, vindt automatisch een reset-bewerking plaats die niet als extra bewerking meetelt. Anders telt elke next- en elke reset-bewerking als 1 bewerking.
Beschrijf hoe je de iterator-bewerkingen implementeert zodat de volgende stelling geldt. Bewijs dat jouw implementatie inderdaad deze eigenschap heeft:
Stelling: *Een reeks van $n > 1$ bewerkingen op een initieel lege semi-splay boom die toevoeg-, verwijder-, opzoek- en iteratorbewerkingen heeft, heeft een geamortiseerde complexiteit van $O(n \log n)$, dus is de gemiddelde kost $O(\log n)$ per bewerking.*

4 Wachtlijnen

In dit hoofdstuk zullen jullie alternatieve implementaties van wachtlijnen zien. Jullie zouden die niet gewoon moeten *aanvaarden* als iets dat jullie *moeten leren* maar vergelijken met implementaties die jullie al hebben gezien – bv. de binaire hoop (binary heap) – en jullie afvragen **waarom** jullie nieuwe implementaties zien: wat zijn de voordelen, de nadelen en de nieuwe ideeën?
Wij zullen hier altijd de klemtoon leggen op de *kleinste* elementen. De bewerkingen zullen dus zo zijn dat de kleinste elementen bijzonder efficiënt gevonden en/of verwijderd kunnen worden. Het is duidelijk dat je helemaal analoog kan werken als het belangrijk is de grootste elementen snel te vinden.

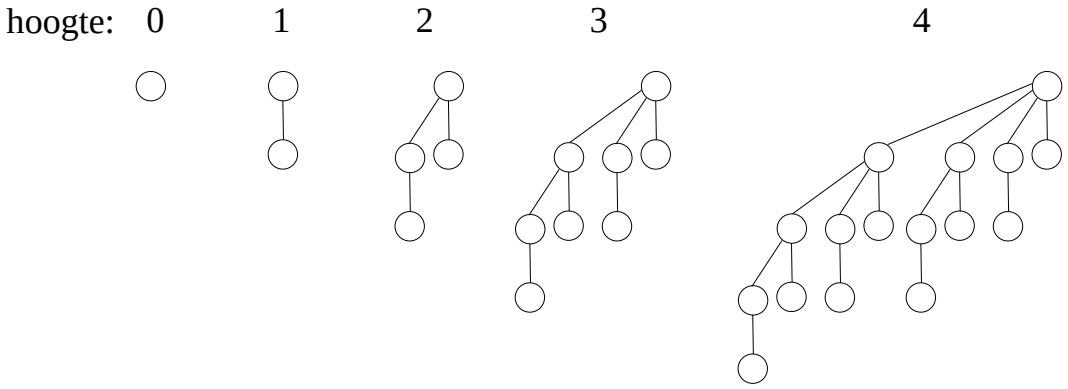
4.1 Binomiale wachtlijnen

Eerst geven wij een inductieve definitie van een binomiale boom:

Definitie 8

- *Een binomiale boom met hoogte* 0 *is gewoon een enkele top.*
- *Een binomiale boom T met hoogte k bestaat uit* 2 *binomiale bomen T_1, T_2 met hoogte $k-1$. De wortel van T is de wortel van T_2 en de wortel van T_1 is een nieuw kind van de wortel van T_2. De kinderen hebben hier geen bepaalde* volgorde.

Een binomiale boom is dus geen binaire boom en ook geen zoekboom. Het is gemakkelijk om te zien (of aan te tonen) dat de wortel van een binomiale boom met hoogte k graad k heeft. Het is even duidelijk dat een binomiale boom met diepte k precies 2^k toppen bevat. In de definitie kunnen we natuurlijk gemakkelijk de rol van T_1 en T_2 verwisselen omdat het allebei binomiale bomen met hoogte $k-1$ zijn en het dus al op voorhand niet duidelijk is welke boom T_1 of T_2 genoemd wordt.



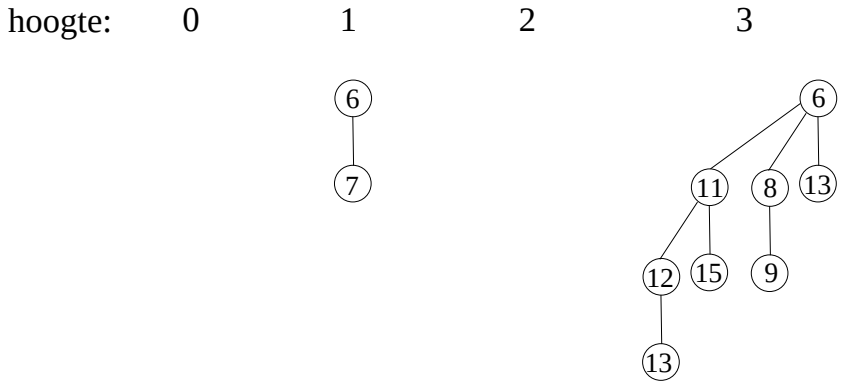
Oefening 41 *Toon expliciet aan dat als je de wortel uit een binomiale boom met hoogte k verwijdert een verzameling van k binomiale bomen overblijft waarin voor elke $i \in \{0, 1, \ldots, k-1\}$ precies één boom aanwezig is.*

Nu kunnen we definiëren wat een binomiale wachtlijn is:

Definitie 9 *Een binomiale prioriteitswachtlijn (soms zeggen wij ook gewoon binomiale wachtlijn) is een verzameling W van binomiale bomen met sleutels in de toppen waarvoor geldt:*

- *Voor elke k zit ten hoogste één boom met hoogte k in W.*
- *De sleutels in de kinderen van een top in één van de binomiale bomen zijn ten minste even groot als de sleutel in de ouder.*

Het is belangrijk dat er meerdere sleutels met dezelfde waarde in een binomiale wachtlijn mogen zitten – anders zijn sommige bewerkingen niet even efficient mogelijk als met dubbele sleutels omdat er dan als een sleutel toegevoegd wordt ook getest moet worden of er al een sleutel met deze waarde in de wachtlijn zit – en dat is duur.



Oefening 42 *Bewijs het volgende lemma:*

Lemma 11 *In een binomiale prioriteitswachtlijn met n toppen is er een boom met diepte k aanwezig als en slechts als de $k$de bit in de binaire voorstelling van n een* 1 *is (waarbij de bit met de laagste waarde nummer* 0 *heeft).*

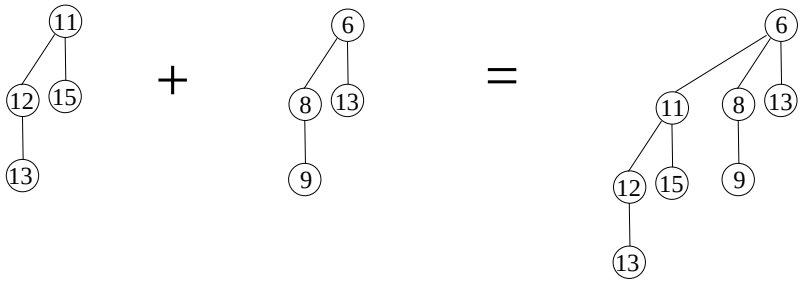
Het bewijs is zeker niet moeilijk en de bedoeling is vooral om te leren dingen die met een beetje nadenken duidelijk lijken *ook zuiver te kunnen bewijzen – om* **echt zeker** *te zijn dat ze kloppen.*

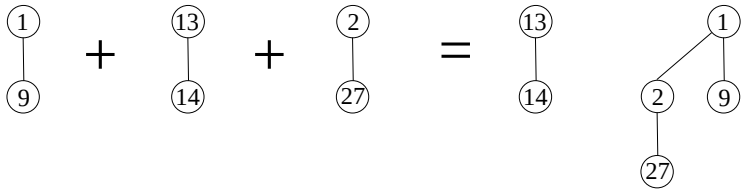
Bewerking: het kleinste element vinden
Hoe kan je het kleinste element in een binomiale prioriteitswachtlijn bepalen? Omdat het kleinste element van elke boom in de wortel zit, moet alleen naar de wortels gekeken worden. Als er een boom met diepte meer dan $\log n$ zou zijn, zouden er meer dan $2^{\log n} = n$ toppen zijn, dus is de grootste diepte $\log n$ en omdat er maar ten hoogste één boom per diepte aanwezig is (beginnend met kleinst mogelijke diepte 0), zijn er dus ten hoogste $(\log n) + 1$ bomen. Wij kunnen het kleinste element dus in tijd $O(\log n)$ vinden.

Alle andere bewerkingen zijn gebaseerd op het mergen van 2 binomiale prioriteitswachtlijnen

Het mergen van binomiale prioriteitswachtlijnen
Het mergen van twee binomiale prioriteitswachtlijnen – dat is het samenvoegen van twee binomiale prioriteitswachtlijnen tot één enkele binomiale prioriteitswachtlijn – is één van de belangrijkste voordelen van binomiale prioriteitswachtlijnen in vergelijking met de binaire hoop. Het is niet alleen een nuttige bewerking op zichzelf maar wij zullen zien dat het ook gebruikt kan worden om het toevoegen van een sleutel en het verwijderen van de kleinste sleutel te implementeren.
De manier van doen lijkt op het optellen van 2 binaire getallen met overdrachten. Wij leggen eerst uit hoe je 0,1,2 of 3 binomiale bomen met dezelfde hoogte k moet mergen.
Als er geen of maar één boom is, moet je niets doen – dat is natuurlijk het gemakkelijkst. De gemergde boom is ofwel leeg ofwel dezelfde boom met hoogte k.
Als je twee bomen T_1 en T_2 hebt, dan kies je de boom met de kleinere wortel als T_2 (kies een arbitraire boom als de wortels gelijk zijn) en bouw je een binomiale boom met hoogte $k+1$ volgens de definitie: je plakt de wortel van de andere boom T_1 aan de wortel van T_2. Het resultaat is dus een binomiale boom met hoogte $k+1$.

Als je drie bomen hebt, kies je een arbitraire van de drie bomen als T_0 en tel je de andere twee op. Het resultaat is dus één boom met hoogte k en één boom met hoogte $k+1$.

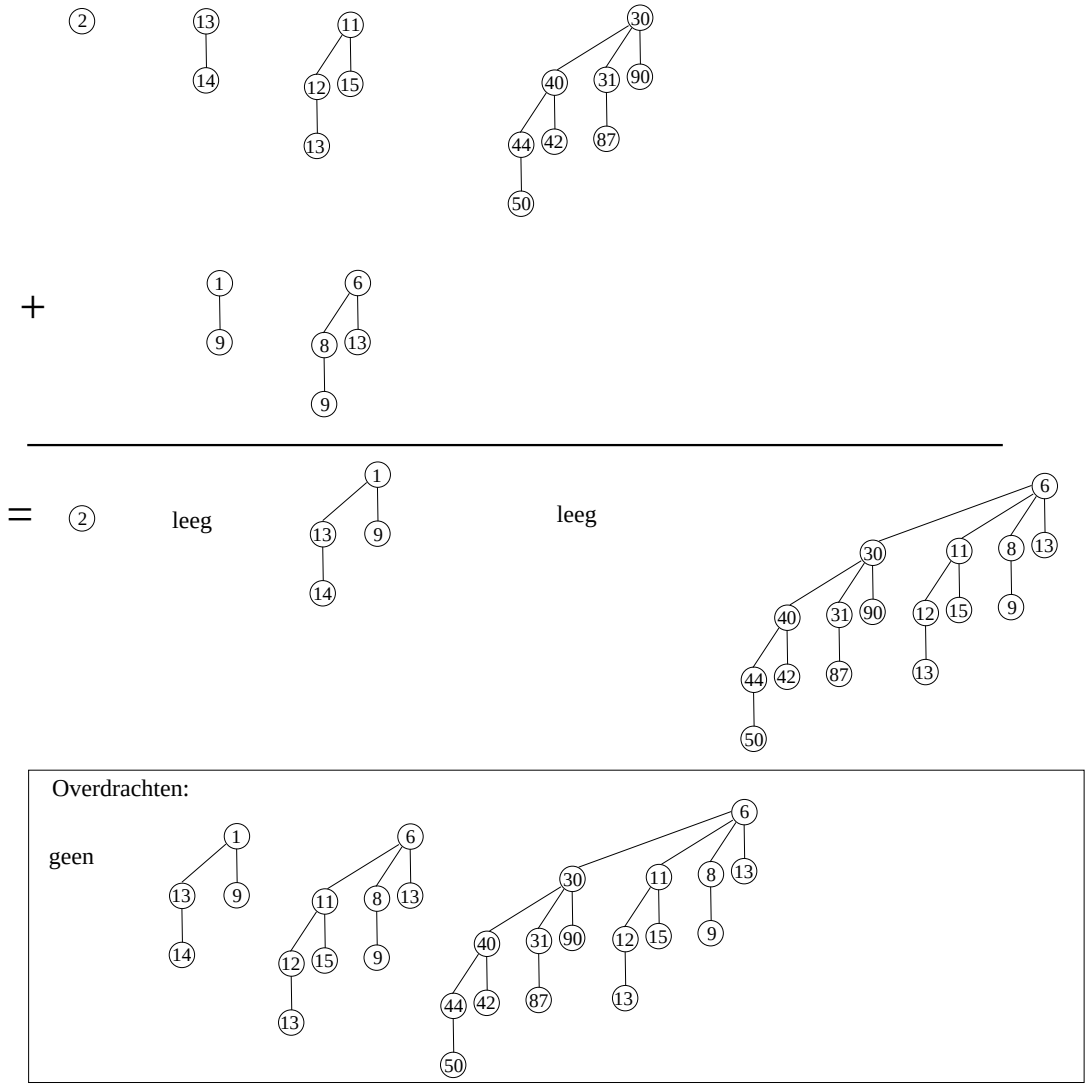




Als wij nu twee binomiale prioriteitswachtlijnen willen mergen dan beginnen wij met de bomen met hoogte 0 en tellen die op (dat zijn er ten hoogste 2) en gaan dan door tot de bomen met de grootste hoogte. Als wij bomen met hoogte k optellen zijn er altijd ten hoogste 3 bomen – ten hoogste twee die van de binomiale prioriteitswachtlijnen afkomstig zijn en ten hoogste één boom overdracht van de vorige stap. Die tellen wij op zodat er geen of één boom met hoogte k is en ten hoogste één boom met hoogte $k+1$ is die wij als overdracht gebruiken voor de volgende stap. Een voorbeeld zie je in Figuur 21.
Het is duidelijk dat het aantal stappen dat nodig is om twee binomiale prioriteitswachtlijnen met b_1 en b_2 bomen te mergen ten hoogste $b_1 + 1$ is als $b_1 \geq b_2$. Een stap is hierbij het mergen van 0,1,2 of 3 bomen en het plaatsen van de nieuwe boom in de binomiale prioriteitswachtlijn. Als er in het resultaat n toppen zitten, is de kost dus $O(\log n)$.

Bewerking: een sleutel toevoegen

Het toevoegen van een sleutel tot een binomiale prioriteitswachtlijn B is nu een gemakkelijke toepassing van mergen: je plaatst de nieuwe sleutel in een – heel gemakkelijke – binomiale prioriteitswachtlijn die maar deze ene sleutel bevat (daarvoor rekenen wij kost 1) en merget die met de al bestaande wachtlijn B. Het toevoegen van een sleutel tot een wachtlijn B met $n > 1$ sleutels heeft dus een kost van $O(\log n)$.



Figuur 21: Een voorbeeld voor het mergen van twee binomiale prioriteitswachtlijnen.

Lemma 12 *Een reeks van n toevoegbewerkingen op een initieel lege binomiale prioriteitswachtlijn W heeft een geamortiseerde kost van $O(n)$, dus gemiddeld een constante kost van $O(1)$ per bewerking.*

Bewijs: Het plaatsen van de nieuwe sleutels in een nieuwe binomiale prioriteitswachtlijn heeft kost 1, wij moeten dus alleen nog het aantal wijzigingen van de bomen in W berekenen. Deze wijzigingen kunnen zijn dat een boom aan een andere boom gehangen wordt of dat een nieuwe boom aangemaakt of verwijderd wordt – dus een pointer naar

de wortel verwijderd of toegevoegd wordt. Zodra je de eerste keer geen overdracht hebt, kan je natuurlijk stoppen met de bewerking. Als een boom b_1 aan een andere boom b_2 wordt gehangen dan moeten bv. de pointer naar de wortel van b_1 verwijderd worden, een nieuwe pointer in de wortel van b_2 geplaatst en misschien de pointer naar de wortel van b_2 op een andere plaats gezet (dat hangt van de exacte inplementatie af).

Voor de analyse vergelijken wij binomiale prioriteitswachtlijnen met ons voorbeeld *binaire teller* uit het vorige hoofdstuk: als er een nieuwe top tot een binomiale prioriteitswachtlijn met m toppen wordt toegevoegd dan is het aantal pointers dat gewijzigd moet worden ten hoogste een constante maal het aantal bits dat in een binaire teller die m voorstelt en waar 1 bijgeteld wordt, gewijzigd wordt. Dit is een gevolg van Lemma 11. Alleen in het gedeelte met de aggregaatmethode hebben wij gebruik gemaakt van het feit dat de teller zo vaak wordt geïncrementeerd dat hij terug 0 is – de andere methoden gebruiken dat niet en tonen dus aan dat elke reeks van n incrementaties op een binaire teller die in het begin 0 is een geamortiseerde kost van $O(n)$ heeft (dus $O(1)$ per bewerking).

Omdat het aantal bomen dat gewijzigd moet worden even groot is als het aantal bits dat in de binaire teller gewijzigd moet worden, volgt het lemma.

■

Bewerking: het kleinste element verwijderen

Stel dat we het kleinste element in een binomiale prioriteitswachtlijn W willen verwijderen. Wij hebben al gezien dat in tijd $O(\log n)$ het kleinste element gevonden kan worden.
Zoals bewezen in Oefening 41 is het resultaat als je de wortel van een binomiale boom met hoogte k verwijdert een verzameling van binomiale bomen waarin voor elke $0 \leq k' < k$ precies één boom met hoogte k' aanwezig is – en dat is natuurlijk een nieuwe binomiale prioriteitswachtlijn..
Als we het kleinste element in W hebben gevonden – stel dat het in boom B zit – kunnen wij dus B uit W verwijderen zodat wij een wachtlijn $W \setminus B$ krijgen. Dan verwijderen wij de wortel van B (constante tijd) en plaatsen alle kinderbomen van de wortel in een nieuwe binomiale prioriteitswachtlijn W_B (dat kan in tijd $O(\log n)$). Nu moeten $W \setminus B$ en W_B nog gemerged worden (dat kan in tijd $O(\log n)$) en het resultaat is een binomiale prioriteitswachtlijn W' die dezelfde elementen bevat als W behalve het kleinste element.

De hele verwijderbewerking vraagt dus tijd $O(\log n)$.

Oefening: Wij hebben niet gezegd op welke manier je de bomen in de wachtlijn bijhoudt. Discussieer over de verschillende manieren om dat te doen en het effect op de efficiëntie van de verschillende bewerkingen.

Oefening 43 *Bouw een binomiale prioriteitswachtlijn W_1 door de elementen $19, 13, 8, 11, 14, 6, 7$ (in deze volgorde) toe te voegen aan een initieel lege binomiale prioriteitswachtlijn . Bouw vervolgens een tweede binomiale prioriteitswachtlijn W_2 door de elementen $1, 5, 22, 8, 14,\ 17, 16, 9, 11, 18, 12$ (in deze volgorde) toe te voegen aan een initieel lege binomiale prioriteitswachtlijn. Merge vervolgens W_1 en W_2 en verwijder ten slotte het kleinste element.*

Oefening 44 *In Lemma 12 hebben wij gezien dat een reeks van n toevoegbewerkingen op een initieel lege binomiale prioriteitswachtlijn een gemiddelde kost van $O(1)$ per bewerking heeft. Maar wat als wij nog andere bewerkingen toelaten? Discussieer en bewijs bovengrenzen voor de geamortiseerde complexiteit voor de volgende rijen van bewerkingen. Bewijs dat jouw grenzen goede bovengrenzen zijn.*

a.) *Je begint helemaal zonder een binomiale prioriteitswachtlijn. Dan heb je n van de volgende bewerkingen:*

- *Aanmaken van een nieuwe binomiale prioriteitswachtlijn met één element (dat is ook mogelijk als er al wachtlijnen bestaan – je werkt dus niet altijd met maar één wachtlijn!). (Kost 1.)*
- *Mergen van twee bestaande binomiale prioriteitswachtlijnen (om het even hoeveel elementen die bevatten). Wij stellen dat de wachtlijn $W_{\leq}$ met minder elementen in de wachtlijn $W_{\geq}$ met meer elementen gemerged wordt en dat de kost hiervan de hoogte van de grootste boom in de nieuwe wachtlijn $W'_{\geq}$ is die gewijzigd werd. (Denk er eens over na waarom dat voor bv. een voorstelling als arraylist de kost goed beschrijft.)*

b.) *Je begint met een lege binomiale prioriteitswachtlijn W. Dan heb je n van de volgende bewerkingen:*

- *Voeg een element tot W toe. Kost: het aantal bomen dat gewijzigd moet worden.*
- *Verwijder het kleinste element uit W. Kost: het aantal bomen in W (voor het zoeken) plus het aantal bomen dat gewijzigd moet worden.*

4.2 Leftist heaps

Een leftist heap is een ander soort heap die het toelaat heaps op een efficiënte manier te mergen. De structuur van een leftist heap is die van een binaire boom.
In binaire zoekbomen was het altijd belangrijk dat we een kleine diepte hadden – dus dat alle paden van de wortel naar een blad zo kort mogelijk waren. De reden is dat wij nooit wisten welke paden gebruikt werden en welke niet. Als wij zouden kunnen garanderen dat alleen zekere paden gebruikt worden, dan zou het voldoende zijn ervoor te zorgen dat deze paden kort zijn en de anderen kunnen ons niet schelen. Dit is het idee dat achter *leftist heaps* staat: gebruik altijd alleen de rechterpaden en zorg ervoor dat die kort zijn!

Definitie 10 *Gegeven een binaire boom B en een top v in B. Dan definiëren wij de null-padlengte $npl(v)$ als het aantal bogen op een kortste pad van v naar beneden (in de richting van de kinderen) tot een top met minder dan 2 kinderen. Als v zelf minder dan 2 kinderen heeft, is dus $npl(v) = 0$. Om dingen gemakkelijker te kunnen opschrijven definiëren wij $npl(\emptyset) = -1$. Op die manier kunnen wij ook van de nullpadlengte van een kind spreken als er geen is.*

Definitie 11 *De rechterpadlengte $rpl(v)$ is het aantal bogen op een langste pad naar beneden dat altijd het rechterkind neemt totdat er geen rechterkind meer is. Dat noemen wij een rechterpad. Wij schrijven ook hier $rpl(\emptyset) = -1$.*

Omdat een rechterpad altijd een pad naar een top met minder dan twee kinderen is, geldt in elke binaire boom B en voor elke top $v \in B$: $\text{npl}(v) \leq \text{rpl}(v)$.

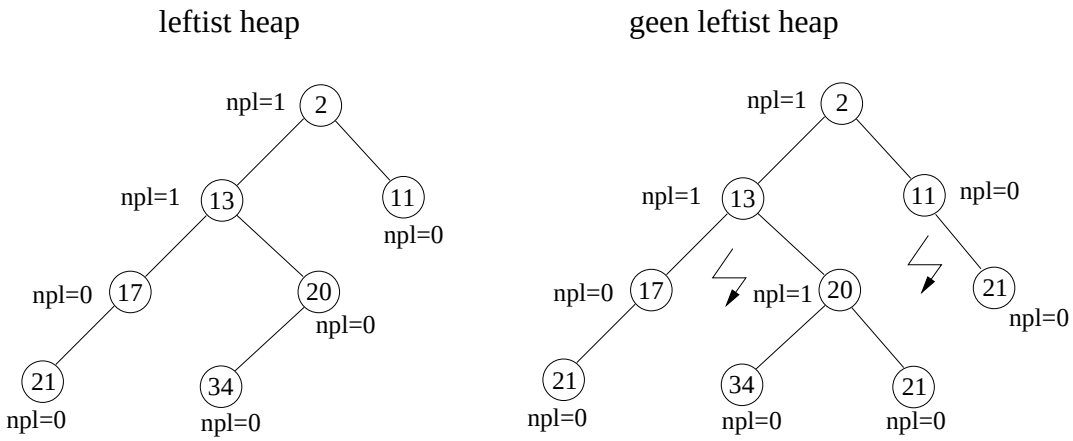
Definitie 12 *Een leftist boom L is een binaire boom waarvoor geldt*

- *Als het linkerkind van $v \in L$ de top l is en het rechterkind r dan geldt $npl(l) \geq npl(r)$*

Omdat de waarde $\text{npl}(v)$ zo belangrijk is in een leftist boom houden wij die voor elke top v bij als wij een leftist boom willen implementeren.

Definitie 13 *Een leftist heap L is een leftist boom die een heap is, dus een leftist boom waarvoor geldt*

- *Voor elke top $v \in L$ geldt dat de sleutels in de kinderen ten minste even groot zijn als de sleutels in v.*



Figuur 22: Een voorbeeld van een leftist heap en van een heap die geen leftist heap is. Er zijn twee plaatsen waar er een probleem is.

Lemma 13 *Een binaire boom B is een leftist boom als en slechts als voor elke top $v \in B$ geldt dat npl(v) = rpl(v).*

Bewijs: Eerst bewijzen wij dat als B een leftist boom is $\text{npl}(v) = \text{rpl}(v)$ geldt:

Voor elke boom geldt $\text{npl}(v) \leq \text{rpl}(v)$. Wij bewijzen $\text{npl}(v) = \text{rpl}(v)$ nu met inductie:

Als v een top is met $\text{rpl}(v) = 0$ dan kan $\text{npl}(v)$ niet kleiner zijn, dus geldt $\text{npl}(v) = \text{rpl}(v)$. Stel nu dat $\text{rpl}(v) > 0$ en dat deze richting van het lemma bewezen is voor alle toppen met een kleinere rechterpad-lengte. Omdat B een leftist boom is, geldt voor het rechterkind r en het linkerkind l dat $\text{npl}(r) \leq \text{npl}(l)$. Het kortste pad van v naar een top met minder dan twee kinderen gaat dus door r en wij hebben dus $\text{npl}(v) = \text{npl}(r) + 1 = \text{rpl}(r) + 1 = \text{rpl}(v)$ (waarbij wij inductie gebruikt hebben). En dat is wat wij wouden bewijzen.

Stel nu dat voor elke top v geldt dat $\text{npl}(v) = \text{rpl}(v)$. Wij moeten bewijzen dat B dan een leftist boom is, dus dat voor elke top v met linkerkind l en rechterkind r geldt dat $\text{npl}(l) \geq \text{npl}(r)$. Omdat $\text{npl}(v) \leq \text{npl}(l) + 1$, geldt $\text{npl}(l) \geq \text{npl}(v) - 1 = \text{rpl}(v) - 1 = \text{rpl}(r) = \text{npl}(r)$.

■

Stelling 14 *Een leftist boom L met wortel w en rpl$(w) = r$ bevat ten minste $2^{r+1} - 1$ toppen.*
De lengte van een rechterpad in een leftist boom L met n toppen is dus ten hoogste $\log(n+1) - 1 \leq \lfloor \log n \rfloor$.

Bewijs: In een leftist boom geldt $\text{npl}(w) = \text{rpl}(w)$. Dat betekent dat alle toppen op afstand ten hoogste $\text{rpl}(w) - 1$ van de wortel 2 kinderen hebben. De boom bevat dus een complete binaire boom met diepte $\text{rpl}(w)$ als deelboom – en deze deelboom heeft al $2^{r+1} - 1$ toppen.

■

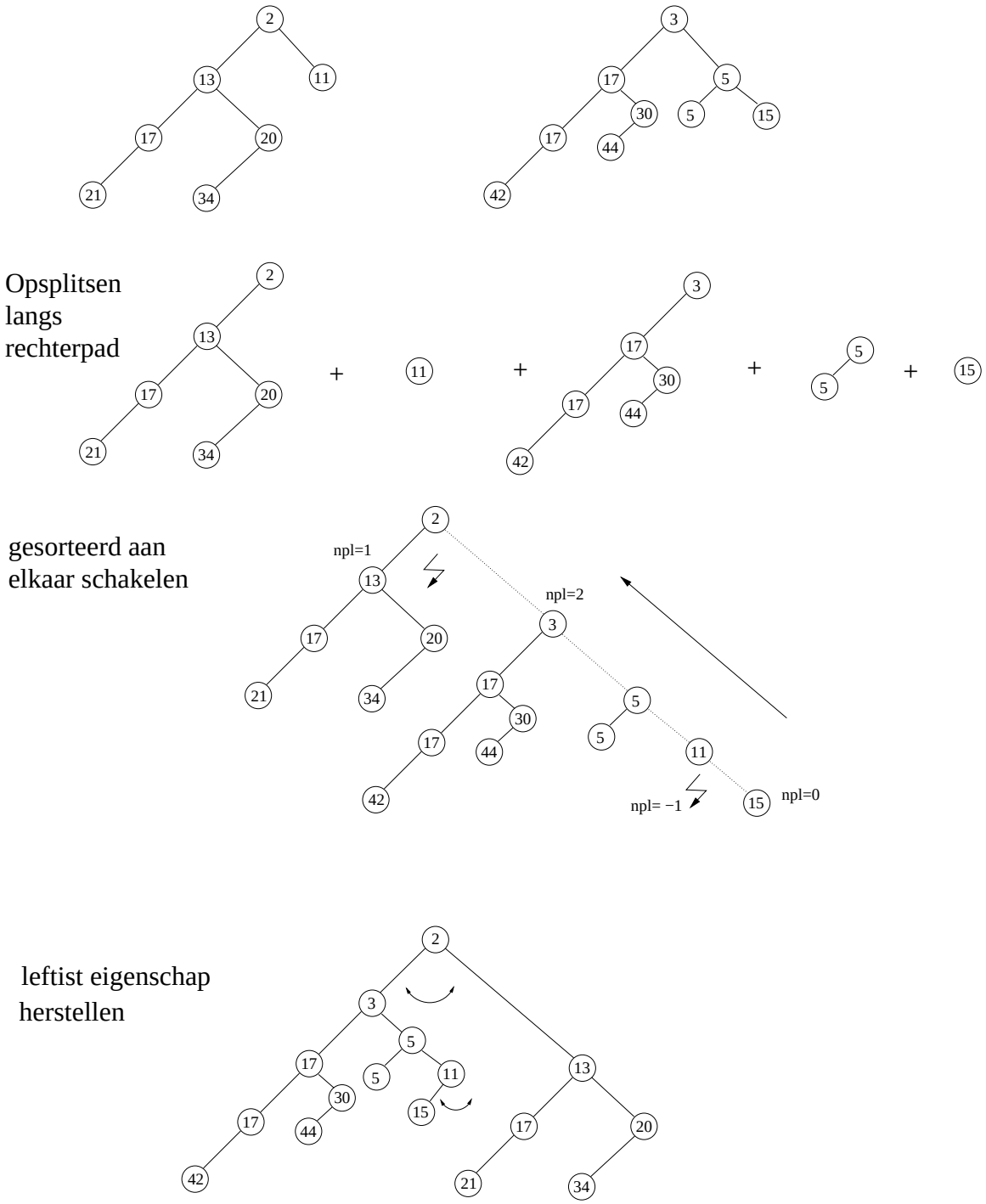
Wij hebben dus een mooie grens voor de lengte van de rechterpaden. In feite zijn die in het slechtste geval nog altijd even kort als bv. de paden van een blad naar de wortel in een binary heap. Voor andere paden geldt dat niet – als de boom bv. alleen maar een lang pad is dat vanaf de wortel altijd naar links loopt dan is dat een heel mooie leftist boom. Het is daarom ook belangrijk dat de bewerkingen op leftist heaps alleen maar met de rechterpaden te maken hebben!

Zoeken:

In een leftist heap zit het kleinste element in de wortel – het kleinste element zoeken is dus geen probleem. Analoog met binomiale prioriteitswachtlijnen zijn alle andere bewerkingen gebaseerd op het mergen van twee leftist heaps.

Mergen:

Hoe twee leftist heaps gemerged moeten worden zie je in Figuur 23.
De toppen op de rechterpaden van de heaps (dat zijn de rechterpaden die aan de wortels vertrekken) samen met hun linkerdeelbomen worden langs de rechterpaden gemerged om een nieuw rechterpad te vormen. Als je het implementeert ga je de oude rechterpaden natuurlijk niet helemaal opsplitsen, maar om te verstaan wat er gebeurt is het misschien het gemakkelijkst te doen alsof de twee leftist heaps langs de rechterpaden opgesplitst worden. Na het opsplitsen worden deze deelbomen gesorteerd volgens de sleutels in hun wortel. Dat kan in tijd $O(\log n)$ gebeuren met n het aantal sleutels in de twee leftist heaps omdat de twee rechterpaden in de heaps al gesorteerd zijn en dus alleen maar gemerged moeten worden (en niet echt opnieuw gesorteerd).
Probleem: Door de deelbomen volgens hun sleutels te sorteren, is het zeker opnieuw een heap, maar niet noodzakelijk met de leftist eigenschap! Maar omdat de kinderbomen alleen maar gewijzigd zijn voor toppen op het nieuwe rechterpad moeten wij alleen op dit pad kijken of de linker- en rechterkinderen nog aan de voorwaarden voldoen. Dit doen wij bv. in een bottom-up manier door eerst voor de laatste top en dan voor zijn ouder, zijn grootouder enzovoort tot wij de wortel bereikt hebben te testen of de top nog aan de leftist eigenschap voldoet: wij kijken of zijn linkerkind l en rechterkind r nog aan de voorwaarde $\text{npl}(l) \geq \text{npl}(r)$ voldoen – en in het geval van niet wisselen wij de kinderen (samen met de bomen die eraan hangen) zodat de top er nadien wel aan voldoet. Natuurlijk moeten in een implementatie alleen maar



Figuur 23: Het mergen van leftist heaps.

twee pointers gewisseld worden. Als wij de leftist eigenschap testen, kunnen wij tegelijk ook de nieuwe waarde van npl() voor de top berekenen: Het is gewoon $\text{npl}(r) + 1$ nadat de leftist eigenschap van de top hersteld is.
Het is inderdaad duidelijk dat de laatste top altijd aan de voorwaarden voldoet, wij kunnen dus met de voorlaatste beginnen.
Na deze herstellingen is de heap terug een leftist heap. Wij hebben dus het volgende lemma:

Lemma 15 *Twee leftist heaps met samen n toppen kunnen in tijd $O(\log n)$ gemerged worden.*

Bewerkingen op een leftist heap : De bewerkingen kunnen nu allemaal gebruik maken van het mergen waardoor ze allemaal in tijd $O(\log n)$ kunnen gebeuren. De waarde $\log n$ is hier een bovengrens voor de lengte van een rechterpad – meestal zal het korter zijn.

- Het toevoegen van een nieuw element tot een leftist heap L gebeurt door het nieuwe element in een leftist heap met maar één element te plaatsen en die dan met L te mergen.
- Het kleinste element wordt verwijderd door de wortel te verwijderen – in een leftist heap zit het kleinste element altijd in de wortel. Het resultaat zijn twee leftist heaps (de linker- en de rechterdeelboom) die dan nog gemerged moeten worden.

Oefening 45 *Bouw een leftist heap L_1 door de elementen $19, 13, 8, 11, 1, 6, 7$ (in deze volgorde) toe te voegen aan een initieel lege heap. Bouw vervolgens een tweede leftist heap L_2 door de elementen $5, 22, 8, 4, 14,\ 17, 16, 9, 11, 18, 12$ (in deze volgorde) toe te voegen aan een initieel lege heap. Merge vervolgens L_1 en L_2 en verwijder tenslotte de wortel.*

Oefening 46

- *Bestaat er voor elke leftist boom B een reeks van toevoegen merge-bewerkingen die toegepast op een initieel lege verzameling van leftist heaps als resultaat een leftist heap L met de structuur van B oplevert? (Precies: waar de L ten gronde liggende boom isomorf is aan B.)*
- *Iets sterker: bestaat er voor elke leftist boom B een reeks van enkel toevoegbewerkingen die toegepast op een initieel lege leftist heap als resultaat een leftist heap L met de structuur van B opleveren? (Precies: waar de L ten gronde liggende boom isomorf is aan B.)*

Oefening 47

- *In de les hebben wij gezien dat een reeks van n toevoegbewerkingen op een initieel lege binomiale heap geamortiseerd tijd $O(n)$ vraagt.*

  Geldt dat ook voor leftist heaps? Indien ja: Bewijs dat. Indien niet: Geef een goede bovengrens voor het aantal stappen (bv. $O(n \log n)$ of $O(n^2)$) en bewijs dat het een goede bovengrens is.

- *Omdat de* stappen *verschillend zijn voor leftist heaps en binomiale prioriteitswachtlijnen tellen wij nu gewoon het aantal vergelijkingen van sleutels. Is er dan een n_0 zodat voor alle $n > n_0$ een rij r_1 van n sleutels bestaat zo dat als die in deze volgorde aan een initieel lege heap worden toegevoegd een leftist heap minder vergelijkingen vraagt dan een binomiale heap? Kan je omgekeerd een dergelijke rij r_2 van sleutels aangeven waarvoor een binomiale heap minder vergelijkingen vraagt?*

Oefening 48 *Geef een rij $s_1, \ldots, s_n$ van sleutels zodat als de sleutels in deze volgorde aan een initieel lege leftist heap worden toegevoegd het resultaat altijd een zo goed mogelijk gebalanceerde binaire boom is.*
Bewijs **expliciet** *dat jouw rij aan deze voorwaarden voldoet.*

Oefening 49 *Het vinden van een zekere sleutel in een heap is niet gemakkelijk. Veronderstel voor het volgende dat je met een zekere top t in de leftist heap al bezig bent – je hoeft die dus niet eerst te zoeken.*
Beschrijf precies hoe je de volgende drie bewerkingen efficiënt kan implementeren. Als er n sleutels in de heap zijn, mogen de bewerkingen maximaal tijd $O(\log n)$ vragen. Na elke bewerking moet het resultaat terug een leftist heap zijn.

remove key: *De top t en zijn sleutel worden uit de heap verwijderd.*

decrease key: *De sleutel in de top t wordt door een kleinere sleutel vervangen.*

increase key: *De sleutel in de top t wordt door een grotere sleutel vervangen.*

De oefening is niet zo gemakkelijk als het misschien op het eerste gezicht lijkt. Kijk bv. naar de situatie waar de sleutel ver in de linkerdeelboom van de wortel zit.

4.3 Skew heaps

Skew heaps lijken op leftist heaps op dezelfde manier als semi-splay-bomen op AVL-bomen lijken: skew heaps zijn zelf-organiserende datastructuren waarvoor wij geen informatie over de structuur bijhouden en waarvoor wij niet kunnen garanderen dat elke bewerking afzonderlijk efficiënt is. Maar ook hier kunnen wij garanderen dat elke (niet te korte) reeks van skew bewerkingen efficiënt kan gebeuren.
De structuureigenschap die wij voor skew heaps eisen, is een heel zwakke vorm van de eigenschap van leftist heaps:

Definitie 14 *Een skew heap is een binaire boom die aan de heap-eigenschap voldoet en waar elke top die een rechterkind heeft ook een linkerkind heeft.*

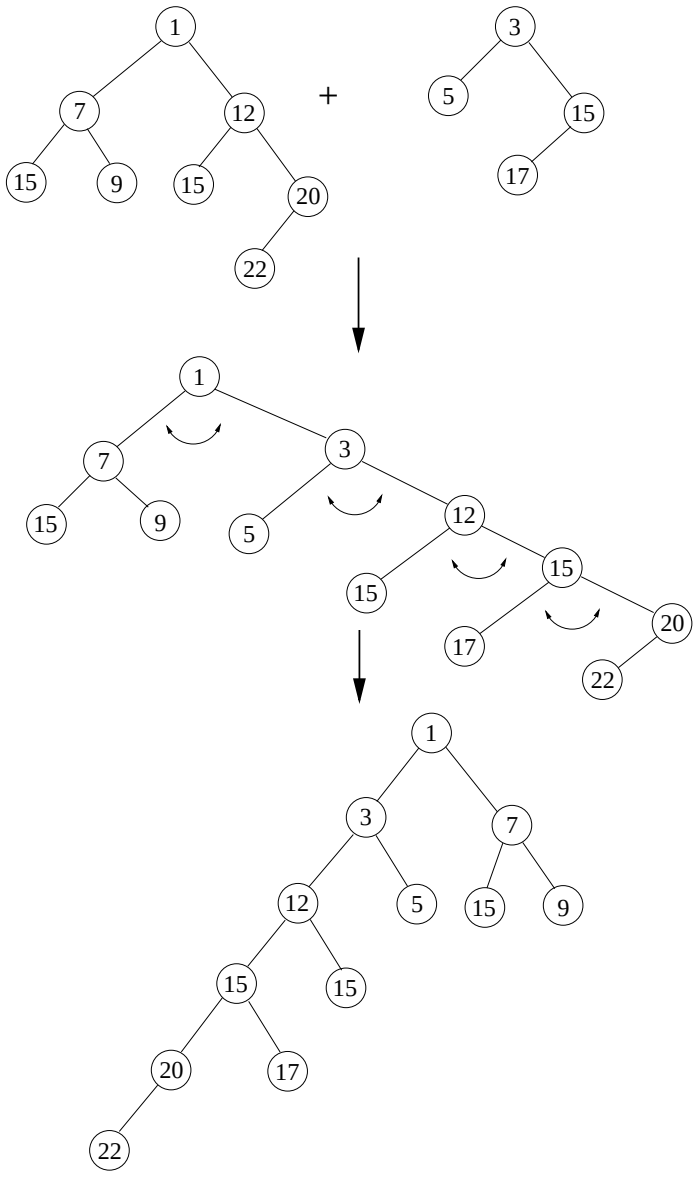
Stel nu dat wij twee skew heaps B_1 en B_2 hebben. Deze mergen wij op een manier die wij een skew merge bewerking noemen. Skew mergen lijkt sterk op de manier waarop leftist heaps gemerged worden en het zorgt ervoor dat als twee skew heaps H_1, H_2 gemerged worden het resultaat H terug een skew heap is:

Definitie 15 *De volgende bewerking op skew heaps noemen wij een skew merge bewerking:*

- *Wij splitsen de twee heaps H_1, H_2 langs de rechterpaden.*
- *Wij vormen met de toppen op de rechterpaden van H_1, H_2 een nieuw rechterpad door de twee paden te mergen. Daarbij zorgen wij ervoor dat het een heap blijft door zo te mergen dat de ouder nooit groter is dan zijn kind.*
- *Voor elke top op het nieuwe rechterpad – behalve de laatste top – verwisselen wij het rechterkind met het linkerkind.*

Het verschil met de merge bewerking voor leftist heaps is dus dat wij er helemaal niet op letten wat de null-padlengten zijn – dat houden wij zelfs niet bij – wij verwisselen gewoon altijd. Wij willen in principe korte rechterpaden – net zo als voor leftist heaps. Daarom lijkt het verstandig dat wij voor de laatste top een uitzondering maken: er is geen rechterkind en dus zou een wissel het rechterpad alleen maar langer kunnen maken. Maar het is ook belangrijk om de skew heap eigenschap te behouden: als wij ook de kinderen van de laatste top zouden wisselen, zouden wij een top met rechterkind maar zonder linkerkind kunnen maken.
Dat het resultaat als je twee skew heaps H_1, H_2 merget terug een skew heap is, kan je bewijzen door naar de toppen te kijken die in de gemergde heap H voor

het wisselen op het rechterpad zitten. Voor alle andere toppen veranderen de kinderen niet – als H_1, H_2 skew heaps waren geldt de skew heap eigenschap dus achteraf voor deze toppen nog steeds. Maar alle toppen op dat rechterpad (behalve de laatste) hebben een rechterkind (anders zou het pad stoppen) – dus na het wisselen zeker een linkerkind – en voldoen dus ook aan de skew heap eigenschap. De laatste top op het rechterpad heeft geen rechterkind en voldoet dus achteraf ook aan de skew heap eigenschap.



Figuur 24: De skew merge bewerking.

De bewerkingen:
Het toevoegen, het verwijderen van het kleinste element en het mergen van twee heaps gebeurt nu op precies dezelfde manier als voor leftist heaps – behalve dat je als je merget altijd een skew merge bewerking gebruikt in plaats van de merge bewerkingen voor leftist heaps.

Oefening 50 *Bouw een skew heap S_1 door de elementen* $19, 13, 8, 11, 1, 6, 7$ *(in deze volgorde) toe te voegen aan een initieel lege heap. Bouw vervolgens een tweede skew heap S_2 door de elementen* $5, 22, 8, 4, 14,\ 17, 16, 9, 11, 18, 12$ *(in deze volgorde) toe te voegen aan een initieel lege heap. Merge vervolgens S_1 en S_2 en verwijder tenslotte de wortel.*
Het zijn opnieuw dezelfde sleutels waarmee je al verschillende heaps hebt gebouwd. Als je de resultaten nog hebt, vergelijk ze met elkaar...

Oefening 51 *Bewijs dat je elke skew heap door een reeks van toevoegbewerkingen en mergebewerkingen op een initieel lege verzameling van skew heaps kan opbouwen.*
Of precies:
Bewijs dat voor elke skew heap H met ten minste 2 toppen 2 kleinere skew heaps H_1, H_2 bestaan, zodat H het resultaat van een skew merge bewerking van H_1, H_2 is.

Het kan gebeuren dat een bewerking op een skew heap met n toppen een kost van $\Theta(n)$ heeft. Maar ook in dit opzicht lijkt het op semi-splay bomen: zodra je naar een voldoende lange reeks van bewerkingen kijkt, is de gemiddelde tijd per bewerking even goed als voor leftist heaps:

Stelling 16 *Een arbitraire rij van m van de volgende bewerkingen op een initieel lege verzameling van skew heaps heeft een geamortiseerde complexiteit van $O(m \log m)$ – dus $O(\log m)$ per bewerking.*
De bewerkingen zijn:

- *een nieuwe skew heap met* 1 *element tot de verzameling toevoegen (kost constant – wij rekenen* 1 *stap)*
- 2 *bestaande skew heaps door middel van de skew merge bewerking mergen (kost lineair in de som van de lengten van de rechterpaden – wij rekenen deze som als kost)*
- *het kleinste element uit een skew heap S in de verzameling verwijderen en de twee delen in plaats van S tot de verzameling toevoegen (zonder ze te mergen) (kost constant – wij rekenen ook hier* 1 *stap)*

Natuurlijk kan de constante bij het verwijderen van het kleinste element iets groter zijn dan bij het toevoegen van een nieuwe heap, maar door het maximum van deze twee als maat voor één stap te kiezen krijgen wij zo zeker een bovengrens voor beide bewerkingen.
Daarbij kan je natuurlijk het aanmaken van een nieuwe skew heap met 1 *element en het onmiddellijke mergen van deze met een al bestaande heap ook als één toevoegoperatie tellen en het verwijderen en mergen van de twee ontstane heaps tot één skew heap ook als één verwijderoperatie. De geamortiseerde complexiteit blijft dezelfde.*

Let op: hierbij wordt wel verondersteld dat je voor het mergen de skew merge bewerkingen gebruikt die je net gezien hebt. Je kan ook merge-bewerkingen bedenken die 2 gegeven skew heaps H_1, H_2 even efficiënt of zelfs efficiënter tot één nieuwe skew heap kunnen mergen – maar die geamortiseerd veel meer tijd vragen...

Bewijs: Wij gebruiken opnieuw de potentiaalmethode. De datastructuur D is hier een verzameling van skew heaps.

Om onze potentiaal te definiëren noemen wij een top *slecht* als er meer toppen in zijn rechterdeelboom zitten dan in zijn linkerdeelboom en anders *goed*. Een top is dus ook goed als er in zijn rechterdeelboom en zijn linkerdeelboom even veel toppen zitten.

De potentiaal definiëren wij nu als

$$\Phi(D) := |\{v \in D | v \text{ is slecht}\}|$$

De potentiaal is dus het aantal slechte toppen in alle skew heaps in de verzameling samen.

Deelresultaat 1: Als er g goede toppen in een rechterpad van een skew heap zitten, bevat de heap ten minste $2^g - 1$ toppen.

Dat volgt gemakkelijk met inductie: Voor $g \in \{0, 1\}$ is het duidelijk, stel dus dat $g > 1$ en v de eerste goede top op het rechterpad (vanuit de wortel) is. Dan kunnen wij voor de rechterdeelboom van v inductie toepassen. Omdat deze deelboom $g-1$ goede toppen op het rechterpad heeft, bevat hij tenminste $2^{g-1} - 1$ toppen. Maar omdat v een goede top is bevat ook zijn linkerdeelboom tenminste $2^{g-1}-1$ toppen. Samen met v zijn dat $2^{g-1} - 1 + 2^{g-1} - 1 + 1 = 2^g - 1$ toppen.

Deelresultaat 2: Als een skew heap met maar één sleutel tot een verzameling D wordt toegevoegd en het resultaat is D' dan geldt $\Phi(D') - \Phi(D) = 0$.

Dat volgt onmiddellijk omdat de nieuwe top goed is.

Deelresultaat 3: Als de wortel van een skew heap S in een verzameling D van skew heaps wordt verwijderd en de nieuwe verzameling met de twee delen van S in plaats van S heet D' dan geldt $\Phi(D') - \Phi(D) \leq 0$.

Dit is gemakkelijk omdat de potentiaal alleen verandert als de verwijderde top slecht was – en dan wordt hij kleiner.

Deelresultaat 4: Als een skew merge bewerking op twee skew heaps S_1, S_2 in een verzameling D wordt toegepast en het resultaat is D' dan geldt: $\Phi(D') - \Phi(D) \leq g - s$ waarbij g het aantal goede toppen en s het aantal slechte toppen is die op het rechterpad van S_1 of op het rechterpad van S_2 liggen.

De eerste stap van de skew merge bewerking is het samenbouwen langs de rechterpaden. Het nieuwe rechterpad heeft dus een lengte van $g + s$. Als je deze heap nog voor het wisselen van de kinderen $\bar{S}$ noemt, dan geldt voor het aantal goede toppen $\bar{g}$ en het aantal slechte toppen $\bar{s}$ op het rechterpad van $\bar{S}$ dat $\bar{s} \geq s$ en $\bar{g} \leq g$: in deze stap kunnen de rechterdeelbomen van de toppen groeien omdat er soms toppen uit de andere heap terechtkomen en de oude toppen in de rechterdeelboom blijven – maar de linkerdeelbomen blijven even groot. Goede toppen kunnen dus slecht worden (omdat er plotseling meer toppen in de rechterdeelboom zitten) maar slechte toppen blijven slecht.

Maar daarna worden alle kinderen langs dit pad gewisseld – behalve voor de laatste top en die is zeker goed. Slechte toppen worden dus zeker goed en goede toppen blijven goed of worden slecht. Laat ons nu naar deze toppen in D' (waar de kinderen gewisseld zijn) kijken. Zij s' hier het aantal toppen die voor het wisselen op het rechterpad zaten en nu slecht zijn. Wij hebben dus $s' \leq \bar{g} \leq g$.

Dus: $\Phi(D') = \Phi(D) + s' - s \leq \Phi(D) + g - s$.

Nu kunnen wij onze tabel maken. Voor het aanmaken van een nieuwe heap en het verwijderen van een wortel rekenen wij een constante kost van 1 stap. Voor het verwijderen van een wortel en het plaatsen van de twee kinderen in de verzameling zou je misschien ook kost 3 kunnen

rekenen, maar omdat 1 hier toch al maar staat voor *constante kost* $O(1)$ zou dat niet echt een verschil zijn. Het mergen is even duur als het aantal toppen op de rechterpaden waarnaar gekeken moet worden. In Deelresultaat 4 is dat precies $g + s$.

	echte kost	gewijzigde kost
één nieuwe skew heap met 1 element toevoegen	1	$1 + \Phi(D') - \Phi(D)$ $= 1$ (Deelresultaat 2)
de wortel van één skew heap verwijderen	1	$1 + \Phi(D') - \Phi(D)$ ≤ 1 (Deelresultaat 3)
twee skew heaps in de verzameling mergen	som van de lengten van de deelpaden: $g + s$	$g + s + \Phi(D') - \Phi(D)$ $\leq g + s + (g - s) = 2g$ (Deelresultaat 4)

Hierbij zou het natuurlijk geen verschil maken als je toevoegen en verwijderen van de wortel met een mergebewerking zou **samenvatten** – de kost zou gewoon die van de mergebewerking zijn.

Maar hoe groot is $2g$? Als g_1, g_2 de aantallen goede toppen op de rechterpaden van S_1, S_2 zijn (dus $g = g_1 + g_2$), dan geldt $g_1 \leq \log(|S_1| + 1)$ en $g_2 \leq \log(|S_2| + 1)$ (Deelresultaat 1). Daarbij is $|S_1| + |S_2| \leq m$ als m het aantal bewerkingen is. Wij zouden opnieuw Opmerking 9 kunnen gebruiken, maar wij kunnen ook direct zien dat $|S_1| + 1 \leq m$ en $|S_2| + 1 \leq m$ (omdat beide heaps ten minste 1 element bevatten) en dus

$$g = g_1 + g_2 \leq \log(|S_1| + 1) + \log(|S_2| + 1) \leq 2 \log m$$

Dus is $4 \log m = O(\log m)$ een bovengrens voor de gewijzigde kost en – omdat de potentiaal op het einde ten minste even groot is als in het begin – $O(m \log m)$ een bovengrens voor de geamortiseerde complexiteit van een arbitraire rij van m bewerkingen op een initieel lege verzameling van skew heaps.

Inderdaad hebben wij ook bij ons argument $|S_1| + |S_2| \leq m$ al toegelaten dat wij een nieuwe skew heap aanmaken en onmiddellijk mergen als één bewerking mogen tellen.

■

Ook intuïtief kan je met behulp van dit bewijs zien waarom skew heaps met skew merge bewerkingen goed presteren. *Slechte* toppen heten niet alleen maar zo, ze zijn ook niet echt goed voor de efficiëntie omdat wij met rechterpaden bezig zijn en dus zo weinig mogelijk toppen op de rechterpaden willen hebben. Toppen met grote rechterdeelbomen hebben natuurlijk een grotere kans lange rechterpaden te hebben en zijn dus *slecht*. Een skew merge bewerking maakt nu eerst langs de rechterpaden het aantal slechte toppen groter – en dan worden ze allemaal goede toppen. De goede toppen kunnen ook slecht worden maar hun aantal wordt niet eerst verhoogd – in tegendeel!

Oefening 52 *Waar hebben wij in het bewijs van Stelling 16 gebruikt dat de kinderen van de laatste top niet gewisseld worden?*

Oefening 53 *Gebruik Opmerking 9 om na te gaan of je op deze manier een betere grens kan vinden.*

Oefening 54 *Het is op het eerste gezicht niet nuttig in een skew heap de kinderen van een top op het rechterpad te wisselen als de heap een linkerkind heeft – ten slotte voldoet deze top al aan de gevraagde voorwaarden van een skew heap.*
Bereken de complexiteit van een bewerking in een reeks van m bewerkingen op een initieel lege verzameling van skew heaps in het slechtste geval en de geamortiseerde complexiteit van de hele reeks van bewerkingen als je de gewijzigde bewerking toepast, die alleen maar de kinderen wisselt als er geen linkerkind is.

Oefening 55 *Bediscussieer de verschillende datastructuren die je kent die een heap implementeren. Wat zijn de voordelen en nadelen van de verschillende datastructuren (bv. qua efficiëntie)?*

4.3.1 Skew heaps recursief mergen

Soms vind je in de literatuur meerdere verschillende versies van *dezelfde* bewerking terug. Als je dan wilt nagaan waarom het *dezelfde* bewerkingen zijn, heb je soms problemen – helemaal dezelfde bewerkingen zijn het namelijk niet, maar ze implementeren alleen hetzelfde *basisidee*. Je kan bv. eens een top-down-implementatie van semi-splay opzoeken. Daar wordt ook hetzelfde basisidee geïmplementeerd – maar helemaal identiek aan de bomen die het resultaat zijn van de bottom-up-bewerking die wij hebben gezien, zijn de bomen na de top-down-bewerking niet.
Hier zullen wij eens naar een recursieve top-down-manier kijken om twee skew heaps te mergen. Het idee dat hier ook verwezenlijkt wordt, is *langs het*

merge-pad wisselen. Maar toch zal het resultaat van deze recursieve skew merge bewerking soms verschillend zijn omdat *het merge-pad* een beetje anders is:

Algoritme 3 Skew-merge bewerking op twee skew heaps.

Het wordt verondersteld dat de twee heaps niet leeg zijn.

Merge *(heap1, heap2)*

- **if** *(wortel(heap1) < wortel(heap2))*
  smallheap = heap1 ; largeheap = heap2
  else *smallheap = heap2 ; largeheap = heap1*

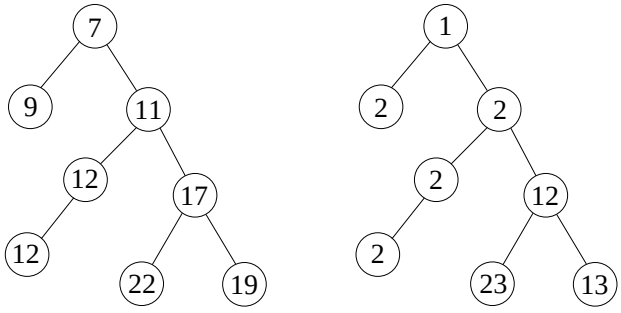
- *wissel de kinderen van de wortel van smallheap*

- **if** *de wortel van smallheap (nu) geen linkerkind heeft:*
  maak largeheap het linkerkind van de wortel.

  else
  maak Merge(linkerkind van smallheap, largeheap)
  het linkerkind van de wortel.

- *return smallheap;*

Oefening 56 *merge de volgende twee skew heaps met de recursieve skew merge bewerking:*



Oefening 57 *Stelling 16 en het bewijs ervan kunnen niet zonder wijzigingen voor de recursieve skew merge bewerkingen gebruikt worden. Welke wijzigingen zijn nodig?*

Oefening 58 *Geef de pseudocode voor een recursieve manier om leftist heaps te mergen. Je kan het best vertrekken van de pseudocode van Algoritme 3.*

5 Verzamelingen

Omdat de datastructuur waarmee wij nu bezig zullen zijn *verzameling* heet, zou het misschien tot verwarring kunnen leiden als wij de *verzameling van alle elementen waarmee wij bezig zijn* ook verzameling noemden. Die noemen wij in dit hoofdstuk dus universum.
Tot nu toe waren wij al bezig met verschillende verzamelingen met een *structuur* die meestal rekening hield met de grootte van de elementen in de verzameling. Ook de bewerkingen waren vaak gebaseerd op de grootte van de elementen die opgeslagen waren (bv. *verwijder het kleinste element*). In dit hoofdstuk houden wij geen rekening met de grootte van de elementen en de bewerkingen die wij efficiënt willen implementeren, zijn vooral elementaire bewerkingen voor verzamelingen.
Stel dus dat een universum gegeven is en dat het bestaat uit getallen $0, \ldots, n$ voor een zekere n. Als het geen getallen zijn (of niet in volgorde vanaf 0), kan je de *echte* elementen gewoon nummers toekennen. Wat zijn de operaties die wij efficiënt willen kunnen doen?

- elementen aan een gegeven verzameling toevoegen
- elementen uit een gegeven verzameling verwijderen
- bepalen of een element in een gegeven verzameling zit
- twee verzamelingen verenigen
- de doorsnede van twee verzamelingen berekenen

Voor niet te grote n en gevallen waar de verzamelingen niet te ijl zijn (de verzamelingen zijn niet te klein in vergelijking met de grootte van het universum) zijn bitvectoren nog altijd een heel efficiënte manier om verzamelingen te implementeren. Het feit dat de nieuwe machines vooral 64-bit integers hebben, maakt het werken met bitvectoren natuurlijk nog efficiënter als $n > 32$.
Wij gebruiken nu het woord woord voor een *long* in java of een *unsigned long int* in C. Stel dat b het aantal bits in een woord is dat je voor de voorstelling van elementen kan gebruiken (dus meestal $b = 64$). In Java moet je wel opletten omdat er een verschil is tussen de aritmetische right shift (die het teken bit op een manier behandelt die hier niet past) en de logische right shift `>>>`. Wij hebben altijd alleen maar logische operatoren nodig maar zullen in de tekst de C-notatie gebruiken.
Een verzameling V is een bitvector – dat is een array van $l = \lceil (n+1)/b \rceil$ woorden – waarin het i-de bit ($0 \leq i < b$) van het j-de woord ($0 \leq j < l$) 1 is als en slechts als element $j * b + i$ in V zit.

Als wij $bit(i)$ voor een woord schrijven dat een 1 op positie $0 \leq i < b$ heeft en alle andere bits zijn 0 (dus $bit(i) = 1 << i$) dan kan je een element k in constante tijd toevoegen, verwijderen of testen of k in de verzameling zit: Stel dat $j = \lfloor k/b \rfloor$ en $i = k \mod b$.

testen of $k \in V$: `(V[j] & bit(i)) != 0`

toevoegen van k tot V: `V[j] = (V[j] | bit(i))`

verwijderen van k uit V: `V[j] = (V[j] & (~bit(i)))`

Dit is natuurlijk heel efficiënt – om het even hoe groot het universum is.
Het berekenen van de unie en de doorsnede van verzamelingen V, W is jammer genoeg niet onafhankelijk van de grootte van het universum. Als het resultaat de verzameling X is, doe je dat bv. met de lus

```
for (i=0; i< l; i++) X[i]= (V[i] | W[i]);
```

resp. voor de doorsnede met

```
for (i=0; i< l; i++) X[i]= (V[i] & W[i]);
```

Dat is lineair in de grootte van het universum, maar omdat het aantal woorden dat je nodig hebt om de verzameling voor te stellen gelijk is aan $l = \lceil (n+1)/b \rceil$ (waarbij dus door b gedeeld wordt) is de constante die impliciet in de uitdrukking $O(n)$ gebruikt wordt (voor niet **te grote** n) heel klein!
Een groter probleem is bv. als je alle elementen van jouw verzameling wil opsommen. Dan moest je *in principe* voor elke $0 \leq i \leq n$ kijken of element i in de verzameling zit en in het geval van een ijle verzameling is dat natuurlijk niet zo efficiënt.... Er waren wel trucjes om dat met een kleine constante te versnellen, maar in principe was dit een bottleneck van deze methode. Intussen hebben alle processoren operaties die dat versnellen – zoals *ffs* (find first set – vind de eerste bit die 1 is), of *clz* (count leading zeros) of *ctz* (count trailing zeros). Natuurlijk is het nog altijd lineair in de grootte van het universum (omdat je naar elk woord moet kijken), maar duidelijk sneller dan als je voor elke bit toetst of het 1 is.
Er is ook veel onderzoek gedaan hoe je sommige operaties efficiënt kan implementeren – zoek eens op het net met het trefwoord *bit twiddling*. Als je bv. wil testen of een verzameling M die als één (unsigned long) integer is voorgesteld ten hoogste één element bevat, dan kan dat met `((M & (M-1)) ==0)`. Nieuwe processoren hebben soms ook daarvoor bitoperaties, die bv. de functie `popcount()` implementeren, die het aantal 1-bits in een computerwoord bepaalt. Bitoperaties zijn dus zeker niet ouderwets, maar cruciaal voor sommige bijzonder snelle programma's.

Oefening 59 *Stel dat jouw universum uit de getallen* $0, \ldots, 15$ *bestaat en dat je* **heel** *vaak voor een getal* $i < 16$ *en als bitvector voorgestelde verzamelingen* V *moet weten wat* het op i volgende element *is – dus het kleinste element in* V *dat groter dan* i *is. Stel een manier voor om dat efficiënt te kunnen beslissen.*

Wat zou je doen als jouw universum duidelijk groter is?

Oefening 60 *Wat berekent de volgende routine:*

```
wat_doe_ik(int m)
{
int i=0;
while (m!=0)
{ m= m & (m-1); i++; }
return i;
}
```

5.1 Union-find algoritmen

Het equivalentieprobleem

Bewerkingen op verzamelingen zijn bijvoorbeeld belangrijk als je met een equivalentierelatie bezig bent en de equivalentieklassen wil berekenen:

Definitie 16 *Een equivalentierelatie* $\equiv$ *op een verzameling* U *is een binaire relatie die voldoet aan de volgende drie eigenschappen:*

(a) reflexiviteit: $u \equiv u$ *voor alle* $u \in U$

(b) symmetrie: als $u \equiv u'$ *dan geldt ook* $u' \equiv u$

(c) transitiviteit: als $u \equiv u'$ *en* $u' \equiv u''$ *dan geldt ook* $u \equiv u''$

Equivalentierelaties zijn op veel vlakken belangrijk.
Voorbeeld 1:
Als U bv. de toppenverzameling van een (niet-gerichte) graaf is en je definieert voor toppen $u, u' \in U$ dat $u \equiv u'$ als en slechts als je top u' vanuit u kan bereiken, dan is $\equiv$ een equivalentierelatie en de samenhangscomponenten van de graaf zijn de equivalentieklassen.

Voorbeeld 2:
Als U een verzameling van vaten is en voor twee vaten $u, u' \in U$ definieer je dat $u \equiv u'$ als en slechts als u en u' even veel inhoud hebben, dan is $\equiv$ een equivalentierelatie.

Voorbeeld 3: (een beetje vaag en algemeen)
Als U een verzameling van *structuren* is en voor twee structuren $u, u' \in U$ definieer je dat $u \equiv u'$ als en slechts als er een structuurbehoudende afbeelding $u \to u'$ bestaat (dat zou je dan een isomorfisme noemen), dan is $\equiv$ een equivalentierelatie. Een speciaal geval van dit voorbeeld heb je als de structuren grafen zijn en de afbeeldingen grafenisomorfismen (of de structuren groepen en de afbeeldingen groepenisomorfismen of...).

Om equivalentieklassen te berekenen, moet je meestal niet naar alle relaties kijken. Stel bv. dat het universum $\{1, 2, 3, 4, 5, 6\}$ is en de relaties zijn $1 \equiv 2$, $3 \equiv 4$, $5 \equiv 6$, $1 \equiv 3$, $4 \equiv 5$, $1 \equiv 4$, $1 \equiv 5$, $1 \equiv 6$, $2 \equiv 3$, $2 \equiv 4$, $2 \equiv 5$, $2 \equiv 6$, $3 \equiv 5$, $3 \equiv 6$, $4 \equiv 6$. Hierbij hebben wij altijd alleen maar één van de relaties $u \equiv u'$ en $u' \equiv u$ gegeven. Die is dan representatief voor beiden.
Wat zijn de equivalentieklassen? Wij beginnen met 6 verzamelingen waarvan elke verzameling maar 1 element uit ons universum bevat. Elementen zitten in dezelfde verzameling als ze zeker tot dezelfde equivalentieklasse behoren en de reflexiviteit garandeert natuurlijk dat elk element equivalent is met zichzelf.
(1) $\{1\}$ $\{2\}$ $\{3\}$ $\{4\}$ $\{5\}$ $\{6\}$
Volgens de relatie $1 \equiv 2$ behoren 1 en 2 tot dezelfde verzameling – dus
(2) $\{1, 2\}$ $\{3\}$ $\{4\}$ $\{5\}$ $\{6\}$
en analoog leiden ook $3 \equiv 4$ en $5 \equiv 6$ tot het verenigen van verzamelingen:
(3) $\{1, 2\}$ $\{3, 4\}$ $\{5, 6\}$
Als wij nu naar $1 \equiv 3$ kijken, dan zien wij dat 1 en 3 in dezelfde verzameling moeten zitten, wij verenigen dus de verzameling met 1 met de verzameling met 3 – dat dan ook 2 en 4 in dezelfde verzameling zitten is volledig in orde – dat volgt uit de transitiviteit!
(4) $\{1, 2, 3, 4\}$ $\{5, 6\}$
De relatie $4 \equiv 5$ zorgt er nu voor dat wij maar één verzameling hebben:
(5) $\{1, 2, 3, 4, 5, 6\}$
Wij moeten dus helemaal niet naar de andere relaties kijken – deze 5 relaties hebben er al voor gezorgd dat alle elementen in dezelfde verzameling zitten, alle andere relaties zijn dus gewoon het gevolg van transitiviteit. Een verzameling van relaties zodat alle andere relaties het gevolg van transitiviteit zijn noemen wij een genererende verzameling van relaties. Ons doel is altijd de door een gegeven verzameling van relaties gegenereerde equivalentierelatie te berekenen. In de tussenstappen hebben wij ook altijd de partitie van de hele

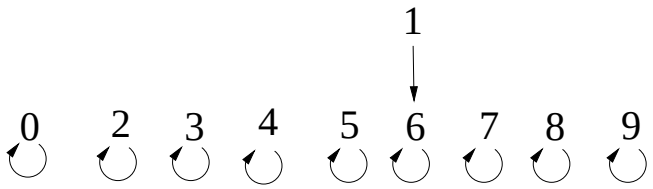
verzameling (dus de equivalentieklassen) die door die relaties gegenereerd wordt die wij al gebruikt hebben. In (4) zijn dat dus de equivalentieklassen van de door $1 \equiv 2$, $3 \equiv 4$, $5 \equiv 6$ en $1 \equiv 3$ gegenereerde equivalentierelatie. Natuurlijk zal het in de meeste gevallen niet zo zijn dat alle elementen van het universum tot dezelfde verzameling behoren zoals in dit voorbeeld!

Twee bewerkingen zijn bijzonder belangrijk als het om de berekening van equivalentieklassen gaat: je moet voor elk element snel kunnen uitvissen in welke verzameling het zit (find) en je moet twee verzamelingen snel kunnen verenigen (union). In dit geval zijn twee verschillende verzamelingen altijd disjunct (elk element zit dus in precies één verzameling) omdat ze equivalentieklassen voorstellen: als de doorsnede niet leeg is, moeten de verzamelingen helemaal gelijk zijn – dat volgt uit de transitiviteit!
In feite kan je dergelijke partities van een universum (dus een verzameling van disjuncte verzamelingen die samen alle elementen van het universum bevatten) gewoon zien als één manier om een equivalentierelatie op dit universum voor te stellen – elke partitie definieert een equivalentierelatie en elke equivalentierelatie een partitie.

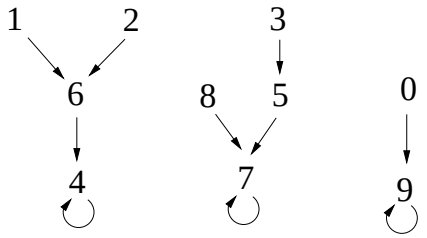
De union-find datastructuur

Het doel is hier de union-bewerking en de find-bewerking voor disjuncte verzamelingen efficiënt te implementeren.
Wij stellen elke verzameling voor als een boom waar de kinderen naar hun ouders wijzen en de wortel naar zichzelf. De wortel stelt de verzameling voor. Als wij equivalentieklassen van elementen uit het universum $\{0, \ldots, n\}$ berekenen en nog helemaal geen relaties gebruikt hebben (dus nog geen verzamelingen verenigd hebben), hebben wij dus $n + 1$ toppen die naar zichzelf wijzen. In het voorbeeld is $n = 9$.

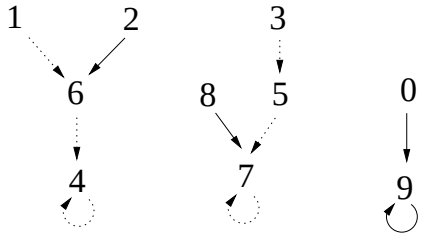
Als een relatie gegeven is, worden eerst de twee verzamelingen bepaald door de pointers te volgen en zo de twee wortels te vinden die de verzamelingen voorstellen. Als in het voorbeeld nu bv. de relatie $1 \equiv 6$ gegeven is, dan is dat gemakkelijk omdat elk element zelf de wortel is. Dan worden de twee verzamelingen verenigd door de pointer van de ene wortel zo te wijzigen dat ze naar de wortel van de andere verzameling wijst:



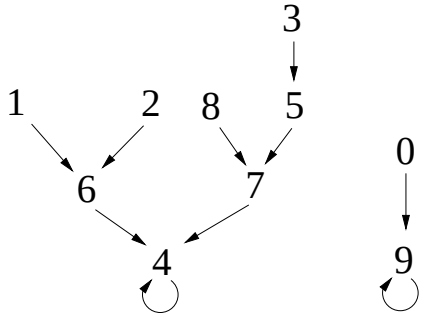
Na sommige unies meer zou het er bv. zo kunnen uitzien:



Als nu bv. de relatie $1 \equiv 3$ gegeven is, loop je langs de met punten getekende pijlen om de wortels van de verzamelingen van 1 en 3 uit te zoeken.



En dan plak je de wortels aan elkaar:



Als jouw universum $1, \ldots, n$ is en de relaties zijn $i \equiv i + 1$ voor $0 \leq i < n$ en je verenigt de verzamelingen door altijd de wortel van de verzameling met element i naar de wortel van de verzameling met element $i + 1$ te laten wijzen dan wordt jouw verzameling op het einde voorgesteld door één pad van lengte n. Dat betekent dat één find- of union-operatie een kost van $O(n)$ kan hebben – en dat is natuurlijk alles behalve goed!

Maar het is ook duidelijk dat het probleem vooral is dat je de grote boom altijd aan de wortel van de kleine plakt. Een eerste verbetering zou dus zeker zijn om in gevallen waar de grootte van de twee bomen verschilt, altijd de wortel van de kleinere boom naar die van de grotere te laten wijzen! Dat wordt union by size genoemd.

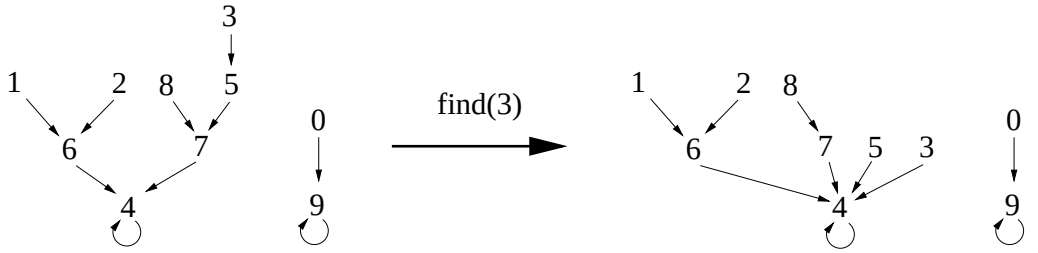
Lemma 17 *Een boom uit een union-find datastructuur die door middel van union by size werd opgebouwd en die m elementen bevat, heeft een diepte van ten hoogste* $\log m$.
Dus hebben een find-bewerking op een verzameling met m toppen en een union bewerking op twee verzamelingen met samen m toppen altijd een kost van $O(\log m)$.

Oefening 61 *Bewijs Lemma 17.*
Toon aan dat de bovengrens uit Lemma 17 een goede bovengrens is – dus dat het kan gebeuren dat de diepte $\log m$ *en dus de kost* $\Omega(\log m)$ *is.*

Oefening 62 *Stel dat T een boom met diepte k is die een verzameling voorstelt en dat T door union by size werd opgebouwd en dat $m(k)$ het minimale aantal toppen is dat in een dergelijke boom aanwezig kan zijn.*

- *Wat is $m(k)$?*
- *Bewijs of geef een tegenvoorbeeld (beide richtingen apart):*
  T bevat $m(k)$ toppen als en slechts als T een binomiale boom is.

Maar het kan inderdaad nog beter door bovendien nog een techniek te gebruiken die path-compression heet. Hierbij wijzig je altijd als je een find-bewerking uitvoert de pointers van alle toppen op het pad – je laat ze naar de wortel van de boom wijzen. Als je dat in onze voorbeeldverzameling doet, gebeurt het volgende:



Hierbij is het natuurlijk belangrijk dat elke keer dat je een relatie $a \equiv b$ evalueert je eerst find-bewerkingen voor a en b moet toepassen. Zo is de geamortiseerde kost van een reeks van m bewerkingen *theoretisch* nog altijd niet constant, maar voor alle praktische doeleinden wel – of precies:

Stelling 18 *(Tarjan)*
Als je union by size en path compression toepast dan vraagt een reeks van m bewerkingen op een universum van grootte n tijd $O(m * \alpha(m, n))$.
Daarbij is $\alpha(m, n)$ gekend als de inverse Ackermann functie, of precies $\alpha(m, n) = min\{i \geq 1 : a(i, \lfloor m/n \rfloor) > \log n\}$

Het bewijs is veel te ingewikkeld voor deze les maar het resultaat is wel belangrijk. De functie $\alpha(n, n)$ is theoretisch niet constant maar groeit zo langzaam, dat in praktische toepassingen de waarde nooit groter dan 5 zal zijn – in de praktijk is het dus even goed als constant – met een kleine constante. De functie groeit bv. trager dan $\log(\log(\ldots(\log n)\ldots)$ – om het even door hoeveel keer log je de eerste punten vervangt...
Dat is omdat de Ackermann functie ongelofelijk snel groeit. De definitie van de Ackermann functie ziet er heel onschuldig uit:

- $a(0, m) := m + 1$
- $a(n + 1, 0) := a(n, 1)$
- $a(n + 1, m + 1) := a(n, a(n + 1, m))$

Dat kan je snel op jouw computer implementeren – maar als je het dan opstart zal je snel verrast zijn. Voor kleine waarden van n hebben wij

- $a(0, m) = m + 1$
- $a(1, m) = m + 2$
- $a(2, m) = 2 \cdot m + 3$
- $a(3, m) = 2^{m+3} - 3$
- $a(4, m) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$ waarbij er $m + 3$ tweeën in de toren van tweeën zitten.

Jullie krijgen al een idee waar dat naartoe gaat... Omdat de Ackermann functie zo snel groeit, groeit de inverse functie zo extreem traag dat wij ze voor praktische doeleinden als constant kunnen beschouwen.
De Ackermann functie zal ook een belangrijke rol spelen in de les *Complexiteit en Berekenbaarheid...*

Union-find en samenhangscomponenten

De samenhang tussen de union-find datastructuur en algoritmen voor het berekenen van samenhangscomponenten is ongeveer dezelfde als tussen zoekbomen en sorteeralgoritmen:
Als je de elementen van het universum interpreteert als de toppen van een graaf en de relaties als bogen in een graaf, dan zijn de equivalentierelaties precies de samenhangscomponenten van de graaf. Die kan je gewoon met diepte-eerst of breedte-eerst bepalen. Het nummer (of de naam) van de component die net berekend wordt, wordt toegekend aan een top op het moment dat je met de top bezig bent. Het opbouwen van de datastructuur kan in lineaire tijd en het bepalen of twee elementen tot dezelfde equivalentieklasse behoren, kan dan in constante tijd – dus theoretisch net iets beter dan union-find waar het (zoals we net hebben gezien) alleen maar *zo goed als constant* per bewerking is en dat ook alleen geamortiseerd en niet in het slechtste geval. Maar als een dergelijke structuur geüpdatet moet worden (bv. door een nieuwe relatie/boog toe te voegen), kan dat best kostelijk zijn...
Dat lijkt op zoekbomen waar je in principe ook een voordeel hebt als je alle elementen al op voorhand kent. Dan kan je ze sorteren (in tijd $O(n \log n)$) en achteraf logaritmisch zoeken in tijd $O(\log n)$ met een heel goede constante. Je hebt het voordeel voor het geheugenverbruik dat je geen extra geheugen nodig hebt voor pointers of balanceringsinformatie en het voordeel voor de cache dat de elementen dicht bij elkaar zitten. Maar ook hier zou een update (bv. het toevoegen van een nieuw element) best duur kunnen zijn...

Oefening 63 *Het universum is* $\{0, 1, \ldots, 8\}$ *en de gegeven relaties zijn* $0 \equiv 2,\quad 3 \equiv 1,\quad 4 \equiv 6,\ 5 \equiv 7,\ 8 \equiv 4,\ 5 \equiv 1,\quad 7 \equiv 2,\quad 0 \equiv 1$.
Bereken de equivalentieklassen van de gegenereerde equivalentierelatie door union-by-size en path compressie toe te passen. In gevallen waar door deze regels niet vastgelegd wordt welke van de twee toppen de wortel wordt, neem die met de grotere sleutel.

Oefening 64 *Je kan union-by-depth definieren door vast te leggen dat elke keer als twee union-find-datastructuren met verschillende dieptes aan elkaar geplakt moeten worden de boom met kleinere diepte aan die met grotere diepte geplakt wordt.*

- *Bepaal de maximale diepte van een union-by-depth datastructuur met n elementen.*
- *Geef een universum en een reeks van union bewerkingen zodat het resultaat van union-by-size een kleinere diepte heeft dan het resultaat van*

union-by-depth – of bewijs dat een dergelijke reeks niet bestaat. Geef uitleg.

- *Analyseer de kost van een reeks van m bewerkingen voor deze datastructuur (dus union- of find-bewerkingen zonder toepassen van path compression). Bepaal een bovengrens en bewijs dat het een goede bovengrens is.*

6 Dynamisch programmeren

Het idee van dynamisch programmeren kan je kort als volgt beschrijven:

- Analyseer of jouw algoritme sommige tussenresultaten heel vaak gebruikt.
- Als dat zo is, bereken ze niet altijd opnieuw maar houd ze in een tabel bij.

Het lijkt natuurlijk duidelijk dat je hetzelfde resultaat niet altijd opnieuw mag berekenen maar soms let je er gewoon niet op...
Het is vaak zo dat het grote aantal herberekeningen door recursieve oproepen wordt veroorzaakt. De bedoeling is echter niet alleen de herberekeningstijd te vervangen door de tijd voor het opzoeken in een tabel. De bedoeling is **vooral** het aantal keren te verminderen dat de waarden gebruikt worden en dat gebeurt door het vermijden van de recursie die voor het herberekenen wordt gebruikt.

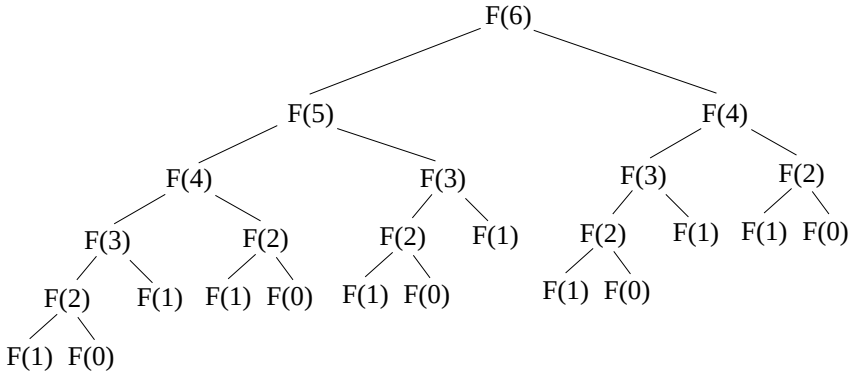
6.1 De Fibonacci getallen berekenen

Het volgende voorbeeld toont vrij goed wat bedoeld is:

Voorbeeld 3 *De Fibonacci getallen $F(n)$ zijn gedefinieerd als $F(0) = 0, F(1) = 1$ en voor $n > 1$ als $F(n) = F(n-1) + F(n-2)$.*
Dat kan je heel gemakkelijk als een recursief programma implementeren en misschien zouden jullie dat ook zo geïmplementeerd hebben:

```
int fibonacci(int n)
{
if (n<2) return n;
else return fibonacci(n-1)+fibonacci(n-2);
}
```

Je moet dan alleen maar op voorhand garanderen dat de routine alleen voor $n \geq 0$ wordt gebruikt.
Als wij naar de recursieve oproepen voor bv. de berekening van $F(6)$ kijken, krijgen we de call-structuur in Figuur 25:
De waarde van $F(5)$ wordt maar 1 keer berekend, de waarde van $F(4)$ twee keer, $F(3)$ drie keer, $F(2)$ vijf keer, $F(1)$ zeven keer en $F(0)$ vijf keer.



Figuur 25: De call-structuur van de recursieve berekening van Fibonacci getallen.

Als je $F(n)$ berekent, zit het **eerste** *blad op afstand $\lfloor n/2 \rfloor$ van de wortel van deze* oproepboom*, alleen in dit deel heb je dus $2^{\lfloor n/2 \rfloor + 1} - 1$ oproepen van de functie – dus duidelijk een exponentieel aantal.*
En omdat je een exponentieel aantal bladeren hebt, zie je ook dat er voor $F(0)$ en $F(1)$ een exponentieel aantal oproepen zijn (en voor $F(2)$, $F(3)$, etc...)

Wij kunnen ook een tabel gebruiken en de waarden van $F()$ vanaf $F(0)$ in de tabel bijhouden:

```
int fibonacci2(int n)

{
  int i;
  int fibonacci[...];

  fibonacci[0]=0;
  fibonacci[1]=1;

  for (i=2; i<=n; i++) fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
  return fibonacci[n];
}
```

Hier zie je dat elke waarde maar één keer wordt berekend en doordat de waarden voor grote n niet herberekend moeten worden, worden de waarden voor kleinere n ook **veel** *minder vaak opgevraagd: elke waarde wordt ten hoogste 2 keer opgevraagd dus is het voordeel niet alleen dat je de waarden kan*

opzoeken in plaats van ze te herberekenen maar dat je veel herberekeningen/opzoekingen vermijdt omdat het grote aantal herberekeningen door de recursie **in** *de herberekeningen werd veroorzaakt.*
Dit algoritme is duidelijk lineair in n*.*

Als je de CPU-tijden van deze twee implementaties vergelijkt dan vraagt de eerste (recursieve) implementatie voor $F(50) = 12586269025$ *ongeveer* 433 *seconden. De tweede is zo snel dat de CPU-tijd nog als* 0 *wordt weergegeven. . .*

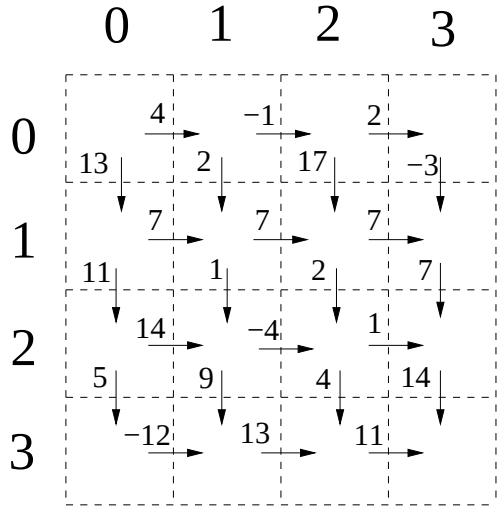
Je zou hier natuurlijk zelfs met maar 2 variabelen in plaats van een tabel kunnen werken maar in ingewikkeldere gevallen is dat niet zo en de bedoeling is natuurlijk vooral het principe te zien.

Oefening 65 *Een definitie die op de Fibonacci getallen lijkt, is die van de faculteit:*

$$f(0) = 1, \qquad f(n) = n * f(n-1)$$

Vergelijk hier de recursieve implementatie en de implementatie die met 0 *begint en een tabel gebruikt. Wat is het belangrijke verschil met de Fibonacci getallen?*

Vooraleer wij naar het iets ingewikkeldere probleem kijken om een optimale volgorde voor het vermenigvuldigen van matrices te vinden, zullen wij eerst nog een ander – iets gemakkelijker – voorbeeld zien om de technieken goed te verstaan:



Op een $n \times n$ schaakbord kan je de vakken met de coordinaten (x, y) labellen waarbij $0 \leq x, y < n$. Elke grens tussen twee buurvakken a en b heeft een waarde $w(a, b)$ die ook negatief kan zijn.
Je kan van een vakje alleen naar rechts of naar beneden gaan – of formeel: Van een vakje (x, y) kan je naar een vakje (x', y') gaan waarbij $x' = x + 1$ en $y' = y$ of $x' = x$ en $y' = y + 1$ (waarbij natuurlijk $0 \leq x', y' < n$).
Gezocht is nu een pad van $(0, 0)$ naar $(n - 1, n - 1)$ zodat de som van de waarden van de grenzen die je hebt overgestoken maximaal is. In de vorige afbeelding is dat $(0, 0) \to (1, 0) \to (2, 0) \to (2, 1) \to (3, 1) \to (3, 2) \to (3, 3)$ met waarde 71. Wij zullen een algoritme geven dat alleen maar de waarde van het pad oplevert, maar door voor elke positie bij te houden waar je voor de beste oplossing vandaan moet komen, kan je met dezelfde kost ook het optimale pad vinden.
De volgende recursieve aanzet is misschien het eerste waaraan je denkt:

```
int get_pad(x,y)
 // geeft het gewicht van een optimaal pad van (0,0) naar
 // (x,y) terug.
{
  if ((x=0) && (y=0)) return 0;
  if (x>0)
       { if (y=0) return (get_pad(x-1,y)+w((x-1,y),(x,y)));
         else
           return max( get_pad(x,y-1)+w((x,y-1),(x,y)),
                       get_pad(x-1,y)+w((x-1,y),(x,y)) );
       }
  else return (get_pad(x,y-1)+w((x,y-1),(x,y)));
}
```

Dat is gemakkelijk om te schrijven, maar je ziet snel dat je heel veel waarden meerdere keren berekent. De waarde van `get_pad(n-2,n-2)` wordt bv. al 2 keer berekend, die van `get_pad(n-3,n-3)` al 6 keer, etc. In feite wordt de waarde van `get_pad(n-1-i,n-1-i)` precies $\frac{(2i)!}{(i!)^2}$ keer berekend. Ook hier kan het veel sneller door gewoon dynamisch programmeren toe te passen en de waarden die meerdere keren worden berekend in een tabel bij te houden. Wij gebruiken een 2-dimensionale tabel `best_pad[x][y]` waarin

wij de waarde van een beste pad naar (x, y) bijhouden:

```
int get_pad_dyn_prog()
 // geeft het gewicht van een optimaal pad van (0,0) naar
 // (n-1,n-1) terug.
{
  best_pad[0][0]=0;
  for (i=1;i<n;i++)
       { best_pad[i][0]=best_pad[i-1][0]+w((i-1,0),(i,0));
         best_pad[0][i]=best_pad[0][i-1]+w((0,i-1),(0,i)); }
  for (i=1;i<n;i++)
    for (j=1;j<n;j++)
      { best_pad[i][j]=
          max( best_pad[i-1][j]+w((i-1,j),(i,j)),
               best_pad[i][j-1]+w((i,j-1),(i,j)) );
      }
  return best_pad[n-1][n-1];
}
```

Het is duidelijk dat het binnenste van de lus constante tijd nodig heeft en het hele programma dus complexiteit $O(n^2)$ heeft. Dat is opnieuw een heel duidelijke verbetering in vergelijking met de recursieve methode! Als je naar de invoerlengte kijkt – dus hoeveel data je moet lezen om het probleem te kunnen oplossen – dan zie je dat dat ook $O(n^2)$ is – het programma is dus lineair in de invoerlengte, wat optimaal is!

Oefening 66 *Toon aan dat voor elk programma dat het probleem met het duurste pad op een $n \times n$ schaakbord in tijd $o(n^2)$ oplost een invoer (dus een probleem) is, waarvoor het algoritme de optimale oplossing niet vindt.*

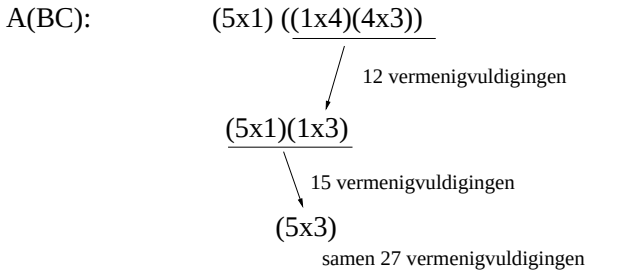
Oefening 67 *Het probleem met het duurste pad op een $n \times n$ schaakbord kan je niet direkt met het Dijkstra algoritme oplossen.*
Vertaal het probleem met het duurste pad op een $n \times n$ schaakbord naar een ander probleem dat je wel met het Dijkstra algoritme kan oplossen. Gezocht is dus een algoritme dat als invoer de invoer voor het $n \times n$ schaakbord probleem heeft en als uitvoer een probleem waarop je het Dijkstra algoritme kan toepassen.
De vertaling moet in tijd $O(n^2)$ mogelijk zijn en uit de oplossing van het vertaalde probleem moet je in constante tijd de oplossing voor het originele probleem kunnen bepalen.

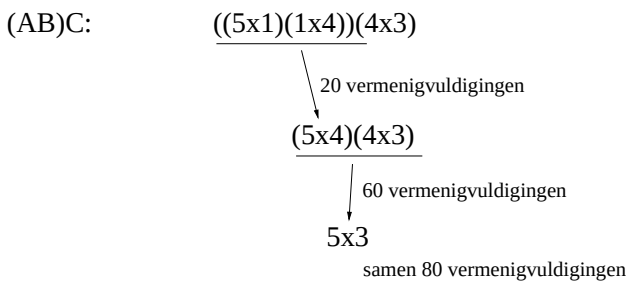
Is het een goed idee op deze manier te werken als de vertaling echt snel is?

Maar nu zullen wij nog een voorbeeld voor dynamisch programmeren zien dat een beetje ingewikkelder is.

6.2 Matrixvermenigvuldiging

Als je twee matrices A, B wil vermeningvuldigen waarbij A een $p \times q$ matrix is en B een $q \times r$ matrix dan vraagt dat $p * q * r$ vermenigvuldigingen van elementen van de matrix (tenminste de normale manier van doen). Ter herinnering: het aantal kolommen van A moet altijd het aantal rijen van B zijn – anders kan je de matrices niet vermenigvuldigen, het is dus geen toeval dat hetzelfde q in $p \times q$ en $q \times r$ opduikt.
Matrixvermeningvuldiging is niet commutatief – je mag dus niet BA berekenen in plaats van AB – ook niet als ze allebei kwadratisch en even groot zijn en beide volgorden mogelijk zijn. Maar matrixvermeningvuldiging is wel associatief: je mag in een langere ketting van vermeningvuldigingen de volgorde van evaluatie veranderen. Voorbeeld:
$(((A \times B) \times C) \times D) = ((A \times B) \times (C \times D))$
De volgorde heeft een invloed op het aantal vermenigvuldigingen dat je moet doen! Voorbeeld:
Stel dat A een 5×1 matrix is, B een 1×4 matrix en C een 4×3 matrix en dat wij ABC willen berekenen.
BC is een 1×3 matrix en wij hebben $1 * 4 * 3$ berekeningen nodig om hem te berekenen





Dat is een vrij groot verschil en we zien dat het – vooral als er meer matrices vermenigvuldigd moeten worden – zeker de moeite is om eerst een optimale volgorde te bepalen voordat wij met de vermenigvuldigingen beginnen.

Als wij $v(i)$ voor het aantal mogelijke evaluatievolgorden bij het product van i matrices schrijven dan hebben wij

$$v(1) = v(2) = 1$$

$$v(n) = \sum_{i=1}^{n-1} (v(i)v(n-i))$$

De beste manier om dat te zien is als volgt: Als de matrices $A_1, \ldots, A_n$ met elkaar vermenigvuldigd moeten worden dan kan je op $n-1$ manieren de **laatste** vermenigvuldiging kiezen – b.v door de laatste matrix van het eerste deel te kiezen. Dat kan ten vroegste matrix A_1 zijn en ten laatste matrix A_{n-1}. Er zijn dus $n-1$ mogelijkheden die te kiezen. Als je ervoor kiest dat matrix A_i de laatste in het eerste gedeelte is, dan heb je $v(i)$ mogelijkheden het product van de matrices in het eerste gedeelte te berekenen en te berekenen en $v(n-i)$ mogelijkheden het product van de matrices in het tweede gedeelte te berekenen. En de mogelijkheden kan je op $v(i)v(n-i)$ manieren met elkaar combineren.
Het resultaat zijn de catalaanse getallen en $v(n+1) = C(n) = \frac{1}{n+1}\binom{2n}{n}$. Het aantal groeit dus exponentieel snel en het wordt al snel vrij duur om alle mogelijke combinaties te testen.

Als `kost(A,B)` de kost is om twee matrices A en B te vermenigvuldigen, dan beschrijft het volgende programma een manier van doen die in principe correct werkt. Daarbij kan je `kost(A,B)` in constante tijd berekenen als de aantallen van rijen en kolommen deel uitmaken van de beschrijving van de matrices en niet eerst berekend moet worden. Ook als in de pseudocode staat `kost((A1x...xAi),(A(i+1)x...xAn))` kan dat in constante tijd. Het is beschrijvend bedoeld en niet zoals een oproep van een functie waar je dan $A_1 \times \cdots \times A_i$ zou berekenen. Om de kost te berekenen moet je allen rekening houden met het aantal rijen van de eerste en kolommen van de laatste matrix in de producten.

```
int get_minkost(A_1, ..., A_n)
{
if (n<=1) return 0;
```

```
min = oneindig;

for (i=1;i<=n-1;i++)
            { buffer = get_minkost(A_1, ..., A_i)
                       + get_minkost(A_{i+1}, ..., A_n)
                       + kost((A_1 × ··· × A_i),(A_{i+1} × ··· × A_n));
              if (buffer<min) min=buffer;
            }
return min;
}
```

Bovenstaand programma zou je de optimale kost geven (en met een kleine wijziging ook de optimale volgorde).
Het is een *verdeel en heers* algoritme dat in principe al efficiënter is dan alle combinaties te testen. Als je nu de recursieve formule voor de kost zou opstellen dan zou die een *plus* in de sommatie hebben waar $v()$ een maal heeft – en dat is natuurlijk al duidelijk beter.
Maar toch bereken je veel waarden meerdere keren in de recursie en het vraagt zeker nog exponentieel veel tijd (vergelijk met de definitie van de Fibonacci getallen)! En dat kunnen wij natuurlijk vervangen door een tabel te gebruiken:
Wij gebruiken een 2-dimensionaal array $minkost[][]$ waarbij $minkost[i][j]$ (in het geval $i \leq j$) staat voor de kost om de vermenigvuldiging $A_i \times A_{i+1} \times \cdots \times A_j$ uit te voeren. In het begin is $minkost[i][j] = 0$ voor alle i, j – of tenminste voor alle $i = j$ (de andere waarden zullen wij niet gebruiken voordat er een nieuwe waarde werd ingevuld).

```
int get_minkost(A_1, ..., A_n)
{

for (lengte=2;lengte<=n;lengte++)
    // in deze lus worden alle kosten voor vermenigvuldigingen van
    // lengte lengte berekend
    for (start=1; start<=n+1-lengte; start++)
    // start is de eerste matrix in de rij van te vermenigvuldigende matrices
    { end=start+lengte-1;
      // end is de laatste in de rij
      minkost[start][end]=oneindig;
      for(j=start;j<end;j++)
```

```
        { buffer= minkost[start][j]+minkost[j+1][end]
                  +kost((A_start × ··· × A_j),(A_j+1 × ··· × A_end));
          if (buffer<minkost[start][end]) minkost[start][end]=buffer;
        }
    }
return minkost[1][n];
}
```

Hier tonen de lussen dat het algoritme duidelijk in tijd $O(n^3)$ draait – dus veel beter dan de vorige algoritmen en zeker voldoende omdat de matrixvermenigvuldigingen achteraf zeker veel duurder zijn.
Het kan inderdaad nog beter, maar de bedoeling is natuurlijk niet het beste algoritme voor dit specifieke probleem te leren kennen maar technieken te verstaan waarmee jullie later **nieuwe** problemen die jullie tegenkomen kunnen oplossen!

Oefening 68 *Wij hebben de pseudocode van dynamisch programmeren altijd op een bottom-up manier beschreven in plaats van een top-down manier zoals de recursieve functies. Was dat belangrijk?*

- *Herschrijf de recursieve functie voor het berekenen van de fibonacci getallen zo dat je een array* `fib[]` *gebruikt waar je een Fibonacci getal invult nadat je het berekend hebt. Bovendien moet de functie ook altijd eerst kijken of* `fib[i]` *al gekend is voordat het berekend wordt. Als dat zo is, wordt natuurlijk het getal uit de tabel gebruikt.*
  Teken de call-structuur voor de oproep voor $n = 6$.
- *Herschrijf de recursieve functie voor het berekenen van de optimale kost voor matrixvermenigvuldigingen zo dat je een array gebruikt waar je gebruikte waarden invult en kijkt of ze al gekend zijn voordat je ze berekent.*

Oefening 69 *De binomiaalcoëfficiënten $C(n,k)$ kan je recursief als volgt definiëren:*

$$\begin{aligned} C(n,0) &= 1\,, \\ C(n,n) &= 1\,, \\ C(n,k) &= C(n-1,k) + C(n-1,k-1)\,, \text{voor } 0 < k < n \end{aligned}$$

(a) *Geef een* **recursief** *algoritme (in pseudocode) dat, voor een gegeven waarde van n en k, de binomiaalcoëfficiënt $C(n,k)$ berekent. Leg uit waarom dit recursieve algoritme inefficiënt is.*

(b) *Geef een algoritme* **met dynamisch programmeren** *(in pseudocode) dat deze definitie gebruikt en voor gegeven waarden van n en k de binomiaalcoëfficiënt $C(n,k)$ berekent. Bespreek de tijds- en geheugencomplexiteit van dit algoritme met dynamisch programmeren.*

Oefening 70 *Stel dat we voor een rij van sleutels $s_1 < s_2 < \cdots < s_n$ al op voorhand weten hoe vaak de sleutels opgezocht gaan worden. Sleutel i wordt g_i keer opgezocht. Het doel is een binaire zoekboom te vinden zodat de opzoekingen zo efficiënt mogelijk gebeuren – dus: waar zo weinig mogelijk toppen bezocht moeten worden. Als op de weg van de wortel naar sleutel i precies $d(i)$ sleutels zitten, is de kost van alle opzoekingen*

$$\sum_{i=1}^{n}(d(i) * g_i)$$

Geef een algoritme met dynamisch programmeren dat de kost van een optimale oplossing bepaalt.
Een tip die gebruikt mag worden (maar niet moet): Stel dat voor $a \leq b$ de boom $B(a,b)$ de boom met de minimale kost voor sleutels $s_a, s_{a+1}, \ldots, s_b$ is en $B(a,b) = \emptyset$ als $b < a$. Dan is de boom met de kleinste kost **en** *sleutel c als wortel de boom die $B(1,c-1)$ als linkerdeelboom en $B(c+1,n)$ als rechterdeelboom heeft. De kost van deze boom is dan $g_c + kost(B(1,c-1)) + kost(B(c+1,n)) + a(1,c-1) + a(c+1,n)$ waarbij*

$$a(i,j) = \sum_{k=i}^{j} g_k$$

dus het aantal keren dat sleutels in deze deelboom opgezocht worden is.

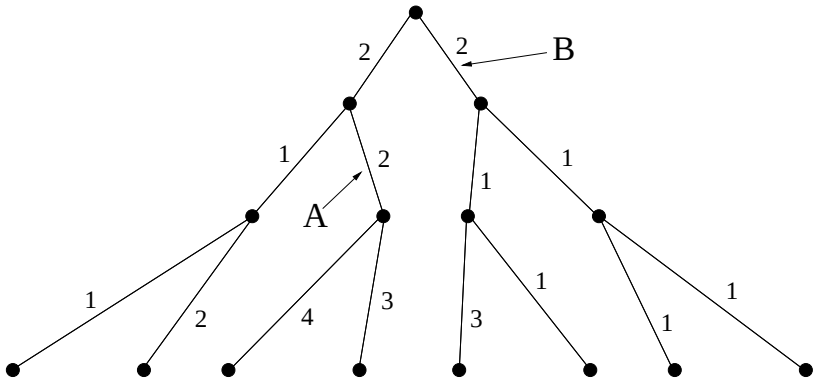
Oefening 71 *Een buis van lengte l mm moet in kleine stukken gesneden worden. Jammer genoeg zijn niet alle lengten nuttig, maar het is wel bekend welke lengten $l_1, \ldots, l_k$ nuttig zijn. De lengten $l_1, \ldots, l_k$ (in mm) zijn veel kleiner dan l, maar zij – en l – zijn allemaal gehele getallen. Natuurlijk eist ook elke doorsnede een beetje ruimte – dat is dus een deel van de buis dat verloren gaat. De vraag is of er een opdeling in stukken van nuttige lengte bestaat zodat de hele buis wordt gebruikt (stel dat elke doorsnede 1 mm breed is). Hierbij bedoelt* opdeling *dat elke snede twee nuttige stukken van elkaar*

scheidt – een snede op het einde van de buis geldt dus niet. Beschrijf een algoritme (bijvoorbeeld pseudocode) met dynamisch programmeren voor dit probleem en toon aan hoe het werkt voor $l = 25, l_1 = 2, l_2 = 3, l_3 = 6$.

7 Branch and bound

Of: vertakken en begrenzen.

Dit deel van de les (tot voor Hoofdstuk 7.2.1) zal vanaf 2014 geen deel meer uitmaken van Datastructuren en Algoritmen 2. Het onderwerp is verhuisd naar Datastructuren en Algoritmen 1. Het blijft wel een deel van de lesnota's als voorbereiding op α–β–snoeien voor die studenten di zich het deel uit Datastructuren en Algoritmen 1 niet meer zo goed kunnen herinneren.
In sectie 6 hebben wij gezien dat je een recursie met veel vertakkingen soms kan vermijden en vervangen door iets dat efficiënter is. Maar dat lukt natuurlijk niet altijd. Soms kan je een vertakking gewoon niet vermijden en dan kan je alleen hopen dat je de vertakking kan begrenzen...



Figuur 26: Een boom waar wij het goedkoopste pad naar een blad zoeken. Twee bogen hebben namen gekregen om er later naar te kunnen verwijzen.

In principe lijkt de situatie als je branch and bound toepast altijd op Figuur 26: je zoekt een goedkoopste pad van de wortel naar een blad in een boom. De kost van een pad is de som van de gewichten van de bogen op het pad. Je begint aan de wortel en doorloopt de hele boom. Dat zou je b.v kunnen doen door de volgende pseudocode functie te gebruiken en met de parameters `top=wortel` en `gewicht=0` en `beste_gewicht=oneindig` op te starten:

```
zoek(top,gewicht)
{
// als de kinderen in volgorde van links naar rechts k[1] tot k[n] zijn
// en de gewichten van de bogen ernaartoe g[1] tot g[n]

if (n=0) en (gewicht<beste_gewicht)
```

```
                    beste_gewicht=gewicht

for (i=1;i<=n;i++)
       zoek(k[i],gewicht+g[i])

}
```

Dit zou een DFS manier zijn om de boom te doorzoeken, maar je kan even goed BFS toepassen.
Inderdaad lijken alle branch and bound situaties op Figuur 26. Het grote verschil is dat de bomen vaak **heel** groot zijn en dat je de bomen niet echt hebt maar dat ze eerder virtueel zijn – zoals de boom in Figuur 25. In de meeste gevallen is de boom die we willen doorzoeken een dergelijke call-structuur van een recursie – maar dat is natuurlijk niet echt een verschil in principe.
Het volledige doorzoeken van de boom is dus lineair in de grootte van de boom. Maar omdat de boom vaak een call-structuur is die exponentieel groot is in vergelijking met de invoerlengte is dat vaak onaanvaardbaar traag.
Aan elke top in Figuur 26 vertak je (dat is dus branch). Maar wat bedoelen wij met *bound*? Stel dat we weten dat alle gewichten groter dan 0 zijn. Als wij dan in onze recursie over boog A naar zijn kind v zijn gegaan, hebben wij het eerste pad al gevonden en `beste_gewicht` is op dat moment 4. Maar ook `gewicht` is al 4 – wij weten dus dat het geen zin heeft om vanuit v door te gaan – als wij vanuit deze top en blad bereiken zal het gewicht groter dan 4 zijn (of precies 4 als v zelf een blad is. Wij moeten dus niet doorgaan met zoeken – en dat noemen wij bound: je doorzoekt sommige takken niet omdat je al weet dat je daar geen betere oplossingen vindt.
Het probleem bij branch and bound is goede en efficiënt berekenbare criteria te vinden om bound toe te passen. De criteria hangen af van de informatie die je over het probleem hebt. Als je in Figuur 26 bv. ook zou weten dat elk pad lengte 3 heeft, zou je na boog B te hebben doorlopen al kunnen stoppen (je hebt al gewicht 2 en krijgt er nog ten minste 2 bij dus gaat het geen verbetering zijn). Te proberen al te voorspellen hoe groot de waarde nog wordt noem je een look-ahead – kijk een stukje vooruit. Als je deze informatie niet hebt, moet je natuurlijk doorgaan.
Dit is een heel triviaal voorbeeld maar het toont reeds alle eigenschappen van branch and bound.
Goede boundingcriteria zijn altijd het resultaat van een zorgvuldige analyse van het specifieke probleem en het is niet mogelijk een algemene manier te beschrijven hoe je die kan vinden. Alles wat wij kunnen doen is een voorbeeld

geven dat hopelijk helpt het principe van branch and bound beter te verstaan en ongeveer een indruk te krijgen hoe je boundingcriteria kan ontwikkelen als je branch and bound later voor een echt probleem zelf moet toepassen...

7.1 Het handelsreizigersprobleem

Het probleem dat wij zullen discussiëren, is het handelsreizigersprobleem. Het handelsreizigersprobleem (in een een beetje andere vorm – als *beslissingsprobleem*) is één van de NP-complete problemen. Dat betekent dat er geen polynomiaal algoritme gekend is (maar dat is maar een klein deel van het verhaal...). Er bestaan heel veel wetenschappelijke artikels daarover hoe je dit probleem het best kan oplossen en wat wij gaan zien is zeker niet de beste manier (die is veel ingewikkelder). Voor ons is het alleen maar een voorbeeld om de technieken te verstaan.

Het handelsreizigersprobleem:

Een handelsreiziger moet een rondreis door n steden $s_1, \ldots, s_n$ maken. Hij kent voor elk paar s_i, s_j van steden de afstand $d(s_i, s_j)$ en zoekt een rondreis door alle steden. Als hij de steden in de volgorde $i_0, \ldots, i_{n-1}, i_0$ bezoekt dan is de lengte van de hele rondreis

$$\sum_{j=0}^{n-1} d(s_{i_j}, s_{i_{(j+1) \bmod n}})$$

waarbij „mod n“ alleen de taak heeft de juiste index i_0 voor de laatste boog $\{s_{i_{n-1}}, s_{i_0}\}$ in de rondreis te krijgen.
Gezocht is nu de kortste rondreis die elke stad precies één keer bezoekt. Omdat het een cykel is, doet het er niet toe waar hij vertrekt.

Er bestaan vrij veel varianten van het handelsreizigersprobleem. In sommige gevallen wordt niet gesteld dat hij tussen alle steden direct kan reizen, in andere varianten mag hij dezelfde stad meerdere keren bezoeken, etc. Maar alle deze varianten lijken heel sterk op elkaar en als je de ene op een goede manier kan oplossen, dan geeft dat altijd ook een goede manier om de andere op te lossen. Als er bv. een verbinding tussen s_i en s_j niet bestaat kan je daaraan gewoon een heel grote afstand toekennen (bv. de som van de afstanden van alle bestaande verbindingen) en als er een rondreis bestaat dan zal een algoritme voor ons (gewijzigde) handelsreizigersprobleem zeker een legale oplossing voor het gewijzigde probleem vinden – dus een oplossing die de niet bestaande bogen niet gebruikt.
Ons model is natuurlijk een gewogen complete graaf. De toppen zijn $s_1, \ldots, s_n$ en het gewicht van een boog $\{s_i, s_j\}$ is $d(s_i, s_j)$.

Het volgende recursieve algoritme zou bv. het gewicht van een goedkoopste rondreis vinden als je het opstart met `zoek(s1,0,1)` en `gewicht_beste_rondreis=oneindig`. Om ook de beste rondreis te hebben en niet alleen het gewicht zou je alleen maar de steden moeten bijhouden.

```
zoek(stad,gewicht,bezochte_steden)
{
// rondreis klaar
if ((bezochte_steden=n)
 { if (gewicht+d(stad,s_1)<gewicht_beste_rondreis))
            gewicht_beste_rondreis=gewicht+d(stad,s_1);
 }
// rondreis nog niet klaar
else
  {
  markeer stad als bezocht
  for (i=2;i<=n;i++)
    if (s_i nog niet bezocht) zoek(s_i,gewicht+d(stad,s_i),bezochte_steden+1)
  markeer stad als nog niet bezocht
  }
}
```

Hoewel het een bijzonder ingewikkeld probleem is, kan je dus **heel** snel een algoritme en zelfs een programma schrijven dat het oplost. Jammer genoeg zal dit programma wel veel tijd vragen als je het draait...
Een principe dat al voor backtracking algoritmen belangrijk is en dus zeker voor branch and bound algoritmen die daarop gebaseerd zijn, is de vertakking laag te houden. Het is natuurlijk veel beter als de vertakking al vanaf het begin laag is! Ons algoritme maakt het pad van s_1 naar `stad` altijd 1 stad langer. De vertakking is het aantal steden waar je vanuit `stad` nog naartoe kan gaan. Het zou dus beter zijn, eerst te kijken aan welk einde van het pad de vertakking lager is en het daar langer te maken – en niet altijd aan de kant van `stad`. Maar daardoor zou het algoritme er iets ingewikkelder uitzien (ten slotte zou de andere kant niet altijd s_1 zijn). Omdat wij hier de klemtoon op de boundingcriteria willen leggen gebruiken wij dus het eenvoudigere algoritme. Maar het is wel belangrijk er altijd aan te denken dat je de vertakking laag moet houden (zie ook de oefeningen).

Oefening 72 • *Hoeveel keren wordt de functie* `zoek()` *opgeroepen als je dit algoritme op een complete graaf met n toppen toepast?*

- *Teken de boom die doorzocht wordt – dus de call-structuur – als je dit algoritme op een complete graaf met 5 toppen toepast.*

Om het algoritme iets efficiënter te maken, moeten wij manieren vinden om bound toe te passen. Daarbij is het belangrijk dat wat je doet om te beslissen of je een tak moet doorzoeken relatief goedkoop te berekenen is – anders is het misschien sneller de tak gewoon te doorzoeken.
Maar natuurlijk zien jullie al een eerste manier om bound toe te passen: in de stap

```
if (s_i nog niet bezocht)
     zoek(s_i,gewicht+d(stad,s_i),bezochte_steden+1)
```

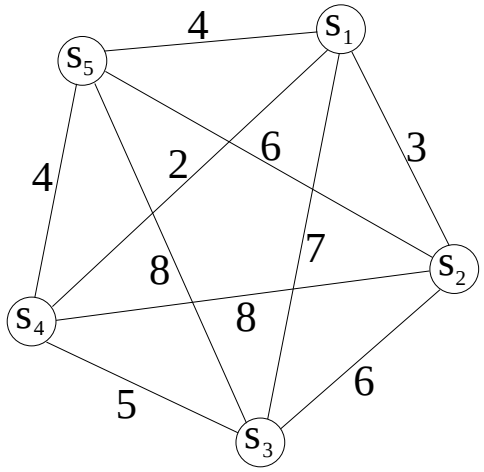
is het natuurlijk veel beter gewoon niet door te gaan als het gewicht al te groot is:

```
if ((s_i nog niet bezocht)
    en (gewicht+d(stad,s_i) < gewicht_beste_rondreis))

         zoek(s_i,gewicht+d(stad,s_i),bezochte_steden+1)
```

Dat is natuurlijk een heel goedkope manier van bound – dus zeker de moeite. De volgende stap zou bv. kunnen zijn dat je al op voorhand bepaalt wat de kleinste van alle afstanden tussen twee steden is (dus $\min\{d(s_i, s_j)|1 \leq i < j \leq n\}$). Dat moet je maar één keer doen en het is lineair in de grootte van de input dus is het ook goedkoop. Als deze kleinste afstand `min_afstand` is dan kan je nog iets tot de lijn toevoegen:

```
if ((s_i nog niet bezocht)
    en (gewicht+d(stad,s_i)+(n-bezochte_steden)*min_afstand
                                  < gewicht_beste_rondreis))

         zoek(s_i,gewicht+d(stad,s_i),bezochte_steden+1)
```

Ook dat is heel goedkoop en toch is het al een kleine look-ahead en kan het helpen al vroeg te herkennen dat sommige takken niet meer doorzocht moeten worden – dus bound toe te passen.
Maar al deze criteria houden niet veel rekening met de verblijvende keuzen voor bogen. Als er bv. één boog duidelijk goedkoper is dan alle anderen en die zit al in het pad of kan niet meer gekozen worden, dan is
`gewicht+d(stad,s_i)+(n-bezochte_steden)*min_afstand`
een heel slechte benadering.



Figuur 27: Een gewogen complete graaf waarin wij de kortste rondreis zoeken.

Een bound criterium dat iets meer op het al gekozen pad en de verblijvende mogelijkheden let, is bv. als volgt:
Voor elke stad s definieer `min_door[s]` als de som van de twee goedkoopste bogen die van s vertrekken. De waarden van `min_door[s]` kunnen één keer op voorhand berekend worden. Als de graaf waarin wij een kortste rondreis zoeken bv. zoals in Figuur 27 is, krijgen wij

```
min_door[s_1]=5
min_door[s_2]=9
min_door[s_3]=11
min_door[s_4]=6
min_door[s_5]=8
```

Dat is dus de goedkoopste manier om in de stad binnen te komen en dan over een andere boog te vertrekken. Stel dat N de verzameling van alle nog niet gekozen steden is en definieer `min_door[N]` als
`min_door[N]` $= \lceil \sum_{s \in N}$ `min_door[s]`$/2 \rceil$
Wij delen door twee omdat je anders gewichten van wegen tussen twee nog niet bezochte steden twee keer telt: over dezelfde boog die je gebruikt om uit één stad te vertrekken kom je een andere stad binnen.
Dus om het even hoe je door de nog niet bezochte steden reist – de kost zal altijd ten minste `min_door[N]` zijn. In feite heb je de eerste en de laatste boog van de verblijvende rondreis zelfs maar half geteld.
In ons voorbeeld gaat in het begin (nog niets gekozen) `min_door[N]` $= \lceil 39/2 \rceil = 20$ zijn – elke rondreis gaat dus zeker ten minste een kost van 20 hebben en wij kunnen de for-lus in sommige gevallen helemaal vermijden:

```
if (gewicht+min_door[N]<gewicht_beste_rondreis)
     { for (i=2;i<=n;i++)
               if (s_i nog niet bezocht) . . . .
     }
```

Dat is nu al iets duurder omdat je (ten minste als je het op de voor de hand liggende manier doet – zie Oefening 73) om `min_door[N]` te berekenen al een lus over toppen hebt, maar het helpt soms ook meer. Als je bv. de rondreis $s_1 \to s_2 \to s_3 \to s_4 \to s_5$ met kost 22 al gevonden hebt (dus `gewicht_beste_rondreis=22`) en de recursie heeft de bogen $s_1 \to s_3 \to s_5$ gekozen dan is `gewicht=15` en `min_door[N]=8` – je kan dus al stoppen. Maar de voordelen zie je natuurlijk niet echt als het over zo kleine voorbeelden gaat.

Oefening 73

- *Bij de berekening van* `min_door[N]` *worden de eerste en de laatste boog op de nog verblijvende rondreis maar half geteld. Kan je* `min_afstand` *gebruiken om de benedengrens nog een beetje beter te maken?*
- *Kan je* `min_door[N]` *zo berekenen dat het in elke recursiestap maar een constante kost heeft en geen kost die lineair in de grootte van N is?*

Oefening 74 *Je kan* `min_door[N]` *voor verzamelingen* `N` *ofwel tijdens de recursie berekenen elke keer dat je het nodig hebt of één keer op voorhand voor alle verzamelingen* `N`*. Wat is beter? Bespreek de voor- en na-delen van deze twee mogelijkheden.*

En zo kan je doorgaan. Deze definitie van `min_door[N]` kan je bv. verbeteren door altijd alleen naar de bogen te kijken die naar toppen leiden die niet al graad 2 in het gekozen pad hebben. Dergelijke bogen kunnen zeker niet meer gekozen worden. Zo zijn de waarden van de gewijzigde `min_door[]` groter en je kan dus vroeger stoppen met een tak doorzoeken. Aan de andere kant is het ook duurder dit nieuwe criterium te berekenen bv. omdat je het nieuwe `min_door[]` niet meer één keer op voorhand kan berekenen en ook een update niet meer constante kost heeft. Je zou `min_door[]` natuurlijk ook alleen dicht bij de wortel (bv. zolang het pad ten hoogste $n/2$ toppen bevat) updaten en dieper in de recursie dan niet meer. Er zijn nog heel veel mogelijkheden om dit branch and bound algoritme te verbeteren.

Maar de bedoeling is hier natuurlijk niet een bijzonder goed algoritme voor het travelling salesman probleem te leren, maar het principe van branch and bound te illustreren en dat is nu hopelijk verstaan.

Sommige basisprincipes voor goede branch and bound algoritmen:

- Probeer de recursie zo te schrijven dat de vertakking zo klein mogelijk is – vooral dicht bij de wortel. (Zie oefening 76.)

- Het probleem is criteria te vinden die aan de ene kant goedkoop te berekenen zijn en aan de andere kant helpen zo vroeg mogelijk (dat is: zo dicht mogelijk bij de wortel) takken te herkennen die geen betere oplossing kunnen bevatten.

- Je kan ook met meerdere boundingcriteria werken – bv. *relatief* dure boundingcriteria in het begin van de recursie en goedkopere als je dieper in de recursie zit. In het begin van de recursie (dus dicht bij de wortel van de call-tree) kan je veel grotere takken vermijden – en dat is zeker de moeite. Bovendien zijn er dicht bij de wortel normaal niet zo veel toppen (oproepen van de recursieve functie) als dieper in de recursie – de bounding criteria worden dus niet zo vaak berekend.

  Dicht bij het einde van de recursie (de bladeren van de call-tree) kan je niet meer zo veel besparen en heb je (normaal) veel meer toppen – hier moeten de bounding criteria dus bijzonder goedkoop zijn!

- Bounding criteria zijn **specifiek voor het probleem** en moeten voor elk probleem aan de hand van het probleem ontwikkeld worden.

Oefening 75 *Teken de call-tree voor het travelling salesman probleem in Figuur 27 als je het bound criterium met* `min_door[N]` *toepast.*

Oefening 76 *Hier gaat het alleen om branch zonder bound – maar je kan er wel voor zorgen dat de vertakking niet te groot wordt...*
Kleuren van grafen
Een consistente kleuring van de toppen van een graaf is een toekenning van een kleur aan elke top zodanig dat twee toppen die door een boog verbonden zijn, een verschillende kleur hebben.
Geef een backtracking algoritme dat voor een graaf met toppen $1, \ldots, n$ *controleert of een gegeven graaf met twee kleuren kan gekleurd worden. Let op de volgorde waarop je de toppen kleurt. Wat is de complexiteit in het slechtste geval als je de toppen in volgorde* $1, \ldots, n$ *kleurt en wat is de complexiteit voor de volgorde die jij hebt gekozen?*
Doe hetzelfde nog eens voor drie kleuren.

Oefening 77 *Gegeven een rij* $g_1, \ldots, g_n$ *van gewichten van pakketten die vervoerd moeten worden. Je hebt identieke vrachtwagens waarop je de pakketten*

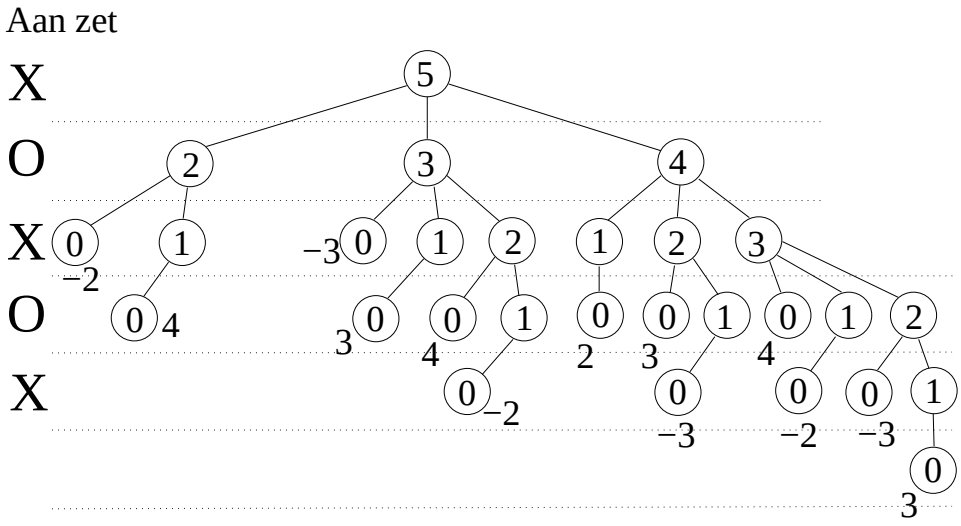
kan plaatsen maar je moet er wel op letten dat de capaciteit C van de vrachtwagens niet overschreden wordt. Het doel is natuurlijk de pakketten op een manier op de vrachtwagens te plaatsen dat zo weinig mogelijk vrachtwagens gebruikt worden.
Geef een branch and bound algoritme voor dit probleem in pseudocode. Gebruik ten minste één bound criterium dat verschillend is van „als je al te veel vrachtwagens gebruikt, zoek dan niet verder maar backtrack“.

7.2 Spelstrategieën

In deze sectie gaat het ook over het doorzoeken van bomen en het begrenzen van de zoektocht maar wij bespreken speciale bomen – namelijk spelbomen. De spelen waarover wij het hier zullen hebben, zijn speciale spelen: er zijn twee spelers die afwisselend een zet doen en de winst van de ene speler is het verlies van de andere. Voor elke zet is er maar een eindig aantal mogelijkheden en de spelen zijn eindig.
Dergelijke spelen kan je beschrijven door spelbomen. De spelbomen zijn in principe de call-structuren als je het spel recursief zou implementeren en aan elke top de mogelijke zetten als vertakking hebt. Natuurlijk moet een speler beginnen. Wij noemen de speler die begint altijd X en de andere altijd O. Wij stellen hier ook dat de spelers altijd de hele spelsituatie (dus ook de al gemaakte keuzes van de andere speler) kennen. Maar wij beginnen het best met een voorbeeld. Het spel is als volgt:

Voorbeeld 4 *Er liggen n 1-cent munten op tafel. De spelers nemen afwisselend $1, 2$ of 3 munten. De speler die de laatste munt neemt, is gewonnen en mag zijn munten houden. De andere speler moet zijn munten terugleggen en bovendien de munten op tafel opvullen tot er terug n munten liggen. Hij is dus evenveel verloren als wat de andere gewonnen heeft.*
In Figuur 28 zie je de spelboom voor $n = 5$. De getallen aan de bladeren (dus waar het spel gedaan is) zijn de winst uit de zicht van X. Als hij verliest is dat een negatief getal. In de toppen staat altijd hoeveel munten er nog op tafel liggen. Als er een boog van de situatie met 3 munten naar de situatie met 1 munt is en X is aan zet, dan betekent dat dat X precies $3 - 1 = 2$ munten heeft genomen.

Spelen zijn vaak niet gewoon *spelletjes*, maar modellen (bv. voor de economie) en dan zijn de uitkomsten natuurlijk heel belangrijk. Wij zullen ons hier met de winst bezig houden.
In veel boeken ga je iets over *het resultaat als alle spelers optimaal spelen* lezen. Maar jammer genoeg is de uitdrukking *optimaal spelen* niet echt ge-



Figuur 28: Een spelboom voor het spel met 5 munten.

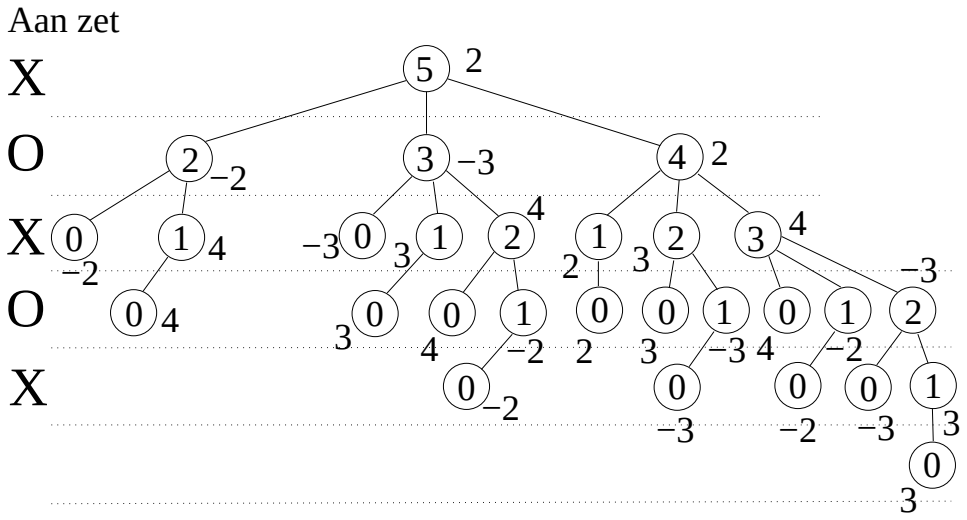
definieerd! Dus is dat niet de beste manier om te beschrijven wat wij als de waarde van een spel willen definiëren.
Als je het informeel wil beschrijven dan zou je kunnen zeggen dat de waarde van een spel het grootste getal w is zodat er – om het even hoe de zetten van O zijn – altijd beslissingen van X zijn die ervoor zorgen dat X ten minste zoveel wint. Maar wij gaan de waarde algoritmisch definiëren. Het is belangrijk erover na te denken dat dat inderdaad beschrijft wat wij willen! Maar om iets te bewijzen zou onze informele beschrijving best iets formeler zijn.

Definitie 17 *Gegeven een spelboom waarbij de bladeren gelabelled zijn met de winst van X. In toppen met een even afstand van de wortel is X aan zet en in toppen met een oneven afstand van de wortel is O aan zet.*
De waarde $w(v)$ van een blad v is dan gedefinieerd als de winst van X als het spel in deze top eindigt.
Nu gaan wij de waarden van de andere toppen recursief definiëren:
Stel dat voor alle toppen t met een maximale afstand van d tot een blad de waarde $w(t)$ al gedefinieerd is en dat v een top met een maximale afstand $d+1$ van een blad is. Dan definiëren wij

$$w(v) = \begin{cases} \min\{w(t) | t \text{ is een kind van } v\} & \text{als } O \text{ in deze top aan zet is.} \\ \max\{w(t) | t \text{ is een kind van } v\} & \text{als } X \text{ in deze top aan zet is.} \end{cases}$$

De waarde van een spel is dan de waarde van de wortel van de spelboom.

In Figuur 29 zien jullie de waarden van alle toppen in ons spelletje met 5 munten. Omdat de waarde 2 is (dus positief) bestaat er voor X dus altijd een mogelijke reeks van zetten die ervoor zorgt dat hij ten minste 2 cent wint.



Figuur 29: De waarde van het spel met 5 munten.

De volgende pseudocode zou dus een manier beschrijven om voor een gegeven spelboom de waarde van alle toppen te berekenen als de functie opgestart wordt met `bereken_waarde(wortel,0)`. Wij stellen dat de winsten van X al als waarden van de bladeren zijn ingevuld en dat de andere waarden ongekend zijn.

```
bereken_waarde(top,afstand_wortel)
// als afstand_wortel even is, is X aan zet anders O
{
if (top is blad) return; // waarde al ingevuld
while (er bestaat kind k met waarde[k]=ongekend)
                  bereken_waarde(k,afstand_wortel+1);

if (afstand_wortel even)
              waarde[top] = max{ waarde[k] | k is kind van top }
else
  waarde[top] = min{ waarde[k] | k is kind van top }
}
```

In feite kan je voor **dit speciale spel** ook op een efficiëntere manier de waarde van de wortel bepalen, maar het is toch hopelijk een goed voorbeeld om de technieken te verstaan.
In de praktijk zullen de spelbomen natuurlijk heel groot zijn en – net zoals voor branch and bound – eerder virtueel zijn dan echt in het geheugen opgeslagen. Wij zijn vooral in de waarde van het spel geïnteresseerd en niet in de waarden van alle toppen omdat als de waarde positief is, is het een *goed spel* voor X – hij kan er door een goede reeks van zetten voor zorgen dat hij winst maakt. Als de waarde negatief is, kan O ervoor zorgen dat O winst maakt en X verliest.
Onze bedoeling is dus de waarde van de wortel op een zo efficiënt mogelijke manier te berekenen – en dat betekent vooral dat wij niet de hele boom willen doorzoeken maar zoveel takken mogelijk willen snoeien.

Oefening 78 *Werk de spelboom uit voor de volgende beginsituatie van drie-op-een-rij.*

X	X	
	O	X
O	O	

Maar: De regels zijn een klein beetje gewijzigd: Je kan niet alleen maar winnen en verliezen maar de winst/het verlies is het aantal stenen die na de laatste zet op het bord aanwezig zijn. Dus hoe later je wint hoe meer je wint. Wat is de waarde van dit spel?

Oefening 79 Een wisselgeldspel
Gegeven een bedrag van b eurocent en een getal m dat het maximale aantal zetten bepaalt. De regels zijn als volgt: In het begin ligt 0 cent op tafel. Speler X begint. Als een speler aan de beurt is en er ligt al een bedrag van $t < b$ cent op tafel, moet hij er precies één munt bijleggen zodanig dat nog steeds ten hoogste een bedrag van b op tafel ligt (1 euro telt natuurlijk als 100 cent, etc). Het spel is gedaan als er b cent op tafel ligt. Als er op dit moment ten hoogste m munten liggen, mag speler X het bedrag houden, anders speler Y. De winst is natuurlijk wat de **andere** *speler op tafel legde.*
Stel nu dat alleen maar munten van 1, 2 en 10 cent toegelaten zijn (anders wordt de vertakking gewoon te groot om het op papier te doen). Wat is de waarde van het spel met $b = 15$ en $m = 5$.

Oefening 80 *Bewijs dat de waarde van een spel inderdaad goed gedefinieerd is. Wat je ook kan doen, is bewijzen dat een programma dat de functie* `bereken_waarde(top,afstand_wortel)` *gebruikt en begint met* `bereken_waarde(wortel,0)` *altijd stopt en dat dan alle waarden berekend zijn.*

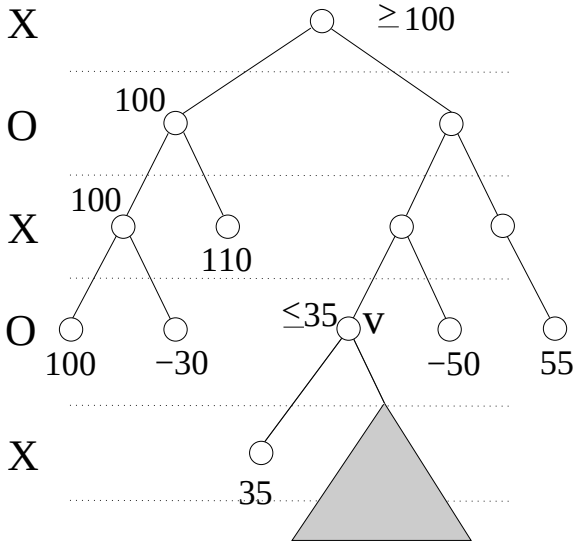
Stel dat je een gerichte graaf hebt en je noemt de toppen bladeren *waarvan geen boog vertrekt en de buren van een top t* kinderen *van t waar een boog vanuit t naartoe gaat. Toppen waar er geen boog binnenkomt noemen wij een* wortel.
Stel dat in het begin aan alle bladeren een waarde toegekend is (aan de andere toppen niet) en dat je maar één wortel hebt. Wat zijn de voorwaarden aan de graaf dat een programma dat de functie `bereken_waarde(top,afstand_wortel)` *gebruikt en begint met* `bereken_waarde(wortel,0)` *stopt?*

7.2.1 (Diep) α–β–snoeien

In dit deel schrijven wij altijd $w(v)$ als wij de waarde van een top v in een spelboom bedoelen. De volgende (gemakkelijke) opmerking kunnen wij gebruiken om de waarde van een spel efficiënter te berekenen:

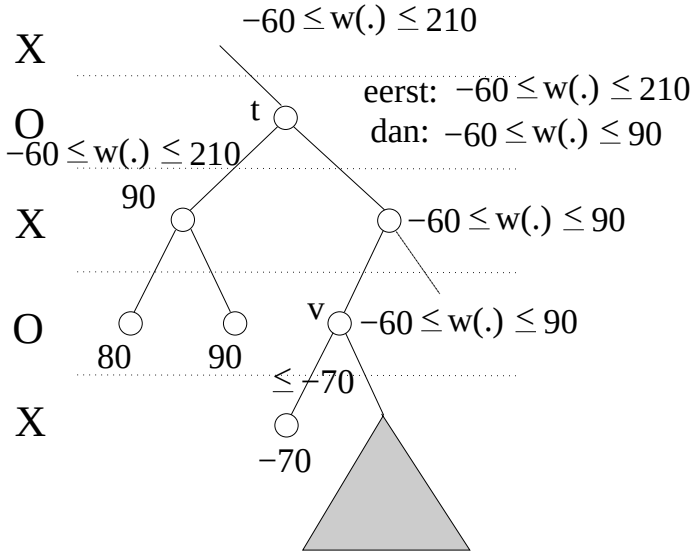
Opmerking 19 *Gegeven een spel met de bijbehorende spelboom. De waarde van het spel – dus de waarde $w(r)$ van de wortel r van de spelboom – is de enige waarde waarvoor er een pad van de wortel naar een blad bestaat zodat alle toppen op het pad dezelfde waarde hebben.*
Een dergelijk pad noemen wij een constant pad.

Het bewijs van deze opmerking is duidelijk: als zo'n pad bestaat, bevat dat de wortel r – de constante waarde is dus $w(r)$. Aan de andere kant heeft elke top een kind met dezelfde waarde als hij – wij vinden dus een pad van de wortel naar een blad waar alle toppen de waarde van het spel hebben.
Als wij alleen maar de waarde van het spel – dus van de wortel van de boom – willen weten dan is het dus niet nodig de waarden van alle toppen te bepalen. Wij moeten alleen maar één pad vinden dat aan de vereisten voldoet. Naar toppen die zeker niet op een dergelijk pad kunnen liggen, moeten wij niet kijken!
In Figuur 30 zien jullie een situatie zoals jullie die tijdens de recursieve berekening van waarden kunnen tegenkomen.
Als wij altijd eerst de linkertak evalueren dan weten wij wanneer wij top v bereiken al dat de wortel r een waarde $w(r) \geq 100$ heeft – ten slotte wordt het maximum gekozen en een kind met deze waarde hebben wij al. Omdat v een top is waar wij het minimum kiezen, weten wij nadat wij de linkertak van v hebben geevalueerd dat $w(v) \leq 35$. Om het even wat de waarden in de



Figuur 30: Een situatie tijdens de recursieve berekening van waarden.

grijze deelboom zijn – de waarden $w(r)$ en $w(v)$ zijn dus zeker verschillend! Dus is v zeker geen top op het pad dat wij zoeken en wij kunnen stoppen met het evalueren van v.



Figuur 31: Een andere situatie tijdens de recursieve berekening van waarden.

In Figuur 31 zien jullie een andere situatie. Hier wordt gesteld dat de grens $-60 \leq w(t) \leq 210$ al gekend is van een top die hoger in de boom zit. Dat is dus een grens die geldt omdat wij **stellen** dat t op een constant pad ligt en

omdat die voor de toppen op het pad naar de wortel geldt. Maar omdat t een top is waar je het minimum kiest, heb je na de evaluatie van het eerste kind ook nog een bovengrens. Die krijg je omdat de waarde van t door het minimum van de waarden van de kinderen wordt bepaald. Maar als een top x op een constant pad ligt dan moeten de grenzen voor de toppen op het pad naar de wortel ook voor x gelden!
Wij hebben dus twee soorten grenzen: aan de ene kant grenzen die door de toppen op het pad naar de wortel vastgelegd worden en gebaseerd zijn op de veronderstelling een constant pad te hebben en aan de andere kant grenzen die door de evaluatie van de kinderen gegeven worden.
Deze grenzen kunnen botsen (zoals bij top v in Figuur 31). Dan weten wij dat de top waarmee wij bezig zijn zeker niet op het pad ligt dat wij zoeken en wij kunnen stoppen met het berekenen van de waarde van deze top. Maar **let op:** dat betekent niet dat de waarde van de top zonder enig belang is – alleen dat de **precieze waarde** niet belangrijk is – het is voldoende te weten dat het buiten de gegeven grenzen ligt (en aan welke kant).
Maar de grenzen kunnen elkaar ook sterker maken – zoals bij top t in Figuur 31. Daar hadden wij in het begin de grenzen $-60 \leq w(t) \leq 210$ maar nadat wij het linkerkind geëvalueerd hebben, weten wij dat (omdat de waarde van de top het minimum van de waarden van de kinderen is) de waarde van deze top ten hoogste 90 is. De nieuwe grenzen voor deze top zijn dus $-60 \leq w(t) \leq 90$. Als wij nu de andere kinderen evalueren kunnen wij al de betere grenzen meegeven. Als deze top op een constant pad ligt dan moeten de nakomelingen van de top die op een constant pad liggen ook aan deze sterkere grenzen voldoen!
De volgende functie in pseudocode berekent de waarde van de wortel. Als aan de hand van de grenzen vastgesteld wordt dat een top onmogelijk op een constant pad kan liggen, wordt niet de echte waarde van de top maar gewoon een getal groter dan of gelijk aan de bovengrens (min-top) of kleiner dan of gelijk aan de benedengrens (max-top) teruggegeven.

```
get_waarde(top,afstand_wortel,ondergrens,bovengrens)

// als afstand_wortel even is, is X aan zet anders O

{
if (top is blad) return waarde(top);

if (afstand_wortel even) // wij moeten het maximum bepalen en de waarde
                         // van een kind geeft een ondergrens
 {
```

```
   for (alle kinderen k)
   { buffer=get_waarde(k,afstand_wortel+1,ondergrens,bovengrens);
     if (buffer>=bovengrens) return buffer;
     if (buffer>ondergrens) ondergrens=buffer;
   }
   return ondergrens;
 }
else // afstand_wortel oneven -- wij moeten het minimum bepalen en
     // de waarde van een kind geeft een bovengrens
 {
   for (alle kinderen k)
   { buffer=get_waarde(k,afstand_wortel+1,ondergrens,bovengrens);
     if (buffer<=ondergrens) return buffer;
     if (buffer<bovengrens) bovengrens=buffer;
   }
   return bovengrens;
 }
}
```

Natuurlijk moeten wij bewijzen dat deze functie de waarde juist berekent – ten slotte worden soms alleen grenzen teruggegeven in plaats van de echte waarden!
Misschien lijkt *„de waarde van toppen waar wij de berekening afbreken doet er toch niet toe“* een overtuigend argument, maar in feite werden met meer overtuigende argumenten al foute veronderstellingen *„bewezen“*... Wij willen dus een bewijs dat iets minder vaag is:

Opmerking 20 *Stel dat t een top met waarde $w(t)$ in een spelboom is waarin de waarden van de bladeren gekend zijn en waarop de functie* `get_waarde()` *met bovengrens b en ondergrens o ($o \leq b$) wordt toegepast.*
Als $o \leq w(t) \leq b$ dan geeft `get_waarde()` *$w(t)$ terug.*
Als $w(t) < o$ geeft `get_waarde()` *een getal $w' \leq o$ terug.*
Als $b < w(t)$ geeft `get_waarde()` *een getal $w' \geq b$ terug.*

Bewijs: Wij gebruiken inductie in de diepte d van de deelboom met wortel t. In het geval van $d = 0$ hebben wij een blad en dat geeft altijd zijn waarde terug. Dat voldoet dus zeker aan deze opmerking.

Stel nu dat $d > 0$ en dat t de wortel van een deelboom met diepte d is en dat `get_waarde()` met bovengrens b en ondergrens o wordt toegepast.

Wij zullen deze stap alleen voor het geval dat t een max-top is expliciet uitwerken, maar het geval van een min-top is volledig analoog.

Wij kijken naar de drie gevallen:

a.) $w(t) < o$**:** Dat betekent dat alle kinderen k van t een waarde $w(k)$ hebben met $w(k) < o$. Dan worden voor alle kinderen dezelfde grenzen toegepast als voor t en elk kind k geeft een getal $r(k) \leq o$ terug (inductie). Dat betekent dat de lus naar alle kinderen kijkt en achteraf o teruggeeft – de opmerking klopt in dit geval dus.

b.) $w(t) > b$**:** Dat betekent dat er een kind k bestaat met $w(k) = w(t) > b$. Als dit kind in de lus niet wordt bereikt dan werd een waarde $w \geq b$ teruggegeven – wat aan de vereisten voldoet. Als k wordt bereikt dan wordt de recursie toegepast met dezelfde bovengrens b (en misschien dezelfde of een andere ondergrens o' met $o \leq o' \leq b$. Volgens inductie voldoet de returnwaarde $r(k)$ dan aan $r(k) \geq b$ en wordt daarna $r(k)$ teruggegeven – wat wij moesten bewijzen.

c.) $o \leq w(t) \leq b$**:** Dat betekent dat behalve in het geval dat $w(t) = b$ waarin b terug wordt gegeven (zoals in geval b.)) naar alle kinderen wordt gekeken omdat geen enkel kind (inductie) een waarde teruggeeft die b of groter is (merk op: de bovengrens wordt niet gewijzigd in de lus). Stel dat k het eerste kind is met waarde $w(t)$. Op het moment dat de lus k bereikt, is dus (inductie) – met o' de ondergrens op dit moment – $o \leq o' \leq w(t) < b$. Volgens inductie geeft k dus de echte waarde terug en is achteraf $o = w(t) < b$. Door de andere kinderen wordt o dan niet meer gewijzigd (inductie) en inderdaad de juiste waarde $w(t)$ teruggegeven.

In elk van de drie gevallen voldoet de teruggegeven waarde dus aan de eisen van de opmerking.

■

Als wij `get_waarde()` met bovengrens ∞ en ondergrens $-\infty$ op de wortel van de boom opstarten (dus met `afstand_wortel=0`) krijgen wij dus de juiste waarde van de wortel en dus van het spel.

Oefening 81 *Bewijs de volgende uitspraken of geef een tegenvoorbeeld:*

- *In een min-top t worden door* `get_waarde()` *altijd getallen r teruggegeven met $r \geq w(t)$ en in een max-top altijd getallen r met $r \leq w(t)$.*
- *Als t een top in een spelboom is die op een constant pad ligt en* `get_waarde(t)` *wordt opgeroepen, dan geeft de functie altijd de juiste waarde terug.*

- *Als t een top in een spelboom is die op een constant pad ligt, dan wordt* `get_waarde(t)` *altijd opgeroepen – gesnoeid wordt alleen bij toppen die niet op een constant pad liggen.*

Oefening 82 *Geeft de volgende functie in pseudocode de waarde van de wortel correct terug? Bewijs dat of geef een tegenvoorbeeld.*
Deze functie moet opgestart worden met `get_NOPwaarde(wortel,0,`$-\infty$`,`∞`)`*. Daarbij gebruiken wij een constante* `NOP`*. Die staat voor* „niet op (constant) pad“.

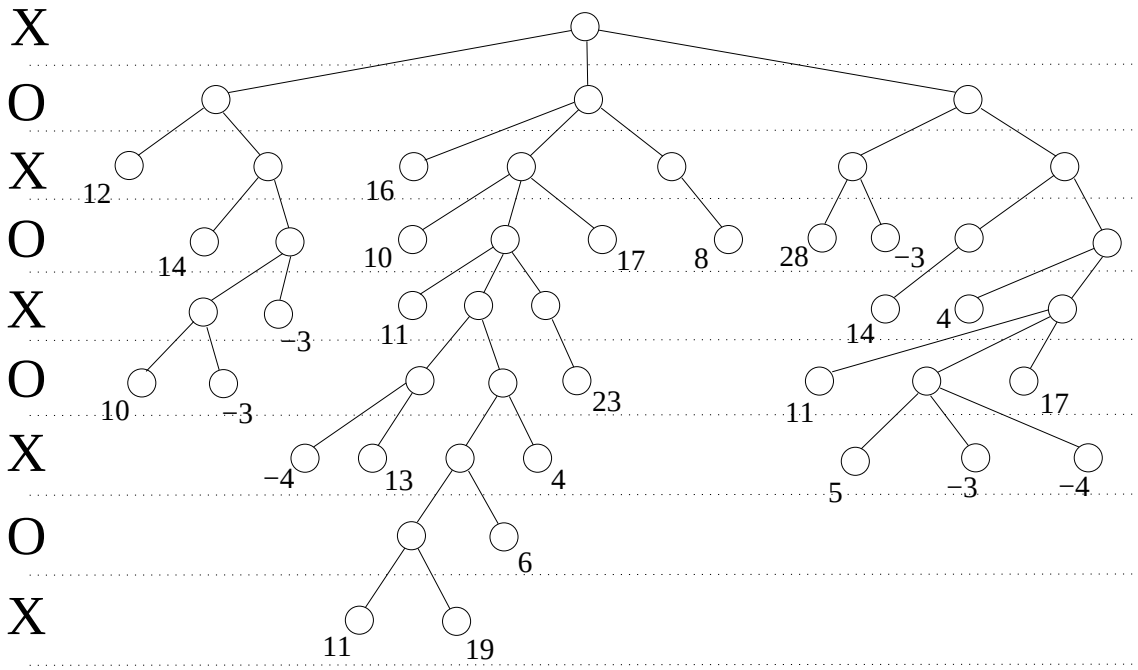
```
get_NOPwaarde(top,afstand_wortel,ondergrens,bovengrens)
// als afstand_wortel even is, is X aan zet anders O
// deze functie geeft ofwel de waarde van de top terug ofwel NOP

{
if (top is blad) return waarde(top);

tempwaarde=NOP;

if (afstand_wortel even) // wij moeten het maximum bepalen en de waarde
                         // van een kind geeft een ondergrens
 {
   for (alle kinderen k)
   { buffer=get_NOPwaarde(k,afstand_wortel+1,ondergrens,bovengrens);
     if (buffer != NOP)
     { if ((tempwaarde=NOP) OF (buffer>tempwaarde))
           { tempwaarde=buffer;
             if (tempwaarde>bovengrens) return NOP;
             if (tempwaarde>ondergrens) ondergrens=tempwaarde;
           }
     }
   }
 }// einde if
else // afstand_wortel oneven -- wij moeten het minimum bepalen en
     // de waarde van een kind geeft een bovengrens
 {
   for (alle kinderen k)
   { buffer=get_NOPwaarde(k,afstand_wortel+1,ondergrens,bovengrens);
     if (buffer != NOP)
     { if ((tempwaarde=NOP) OF (buffer<tempwaarde))
```

```
            { tempwaarde=buffer;
              if (tempwaarde<ondergrens) return NOP;
              if (tempwaarde<bovengrens) bovengrens=tempwaarde;
            }
      }
    }
 }// einde else

return tempwaarde;

}
```

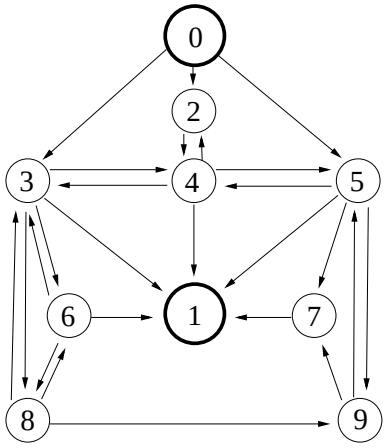
Trucjes

Als je het over heel grote spelen hebt – bv. schaken – is het natuurlijk nog steeds niet mogelijk de spelboom volledig te evalueren – zelfs niet als jouw startpositie in feite een positie is die al meerdere zetten na de echte start is. Voor dergelijke gevallen kunnen heuristieken toegepast worden: ook al zijn de waarden van het spel misschien maar $+1$ of -1 voor winnen en verliezen – je kan gewoon waarden die de *verwachte winst van* X beschrijven op een heuristische manier toekennen aan posities die geen eindposities zijn. In het geval van mogelijke uitkomsten $+1$ en -1 bv. -1 *plus twee keer de kans dat* X *wint*. De kans dat X wint inschatten is natuurlijk alles behalve gemakkelijk. Voor dergelijke heuristieken moet je het specifieke probleem heel goed kennen en er zijn geen gemakkelijke regels die je alleen maar moet volgen om een goed algoritme te krijgen (zoals het heel vaak het geval is...).

Oefening 83 *Bereken de waarde van dit spel. Evalueer altijd eerst de linkertak van een top en schrijf de grenzen aan de top. Pas α–β–snoeien toe.*



Oefening 84 *Een spel op een graaf: De spelers zijn X en O en ze kiezen afwisselend toppen. Elke top kan maar 1 keer gekozen worden. Speler X begint en* **moet** *top nummer 0 kiezen. Als net een top v werd gekozen dan moet de volgende top w een top zijn waarvoor een gerichte boog $v \to w$ bestaat (dus worden in principe gerichte paden gekozen). Het spel is gedaan als top 1 wordt gekozen (dan is de speler gewonnen die deze top heeft gekozen) of als geen top meer gekozen kan worden (dan is het gelijkspel en niemand wint iets).*
In het geval dat een speler wint, is zijn winst de som van de van **hem** *gekozen toppen. De andere speler moet hem deze winst betalen – dus is hij even veel verloren.*
Als in de gegeven graaf de toppen $0, 3, 1$ in deze volgorde worden gekozen, wint X dus $0 + 1 = 1$ cent en als $0, 3, 4, 2$ in deze volgorde worden gekozen is de winst 0 voor beide spelers.



- *Wat is de waarde van dit spel als je het op deze graaf speelt?*
- *Teken de spelboom en pas α-β-snoeien toe. Als je snoeit, geef uitleg waarom dat kan.*
- *Evalueer eerst de tak waar O voor top 2 kiest.*

Oefening 85 *Beschrijf een dynamisch programmeren algoritme dat de waarde van het muntspelletje van Voorbeeld 4 met n munten bepaalt.*

8 Heuristieken en online algoritmen

heuristiek: De leer van het vinden (Grieks: heuriskein=vinden) of de manier via een methodische weg kennis te verwerven.

Ook: dienend om iets te vinden (Grieks: heuretikos=vindingrijk)

Als wij vandaag heuristiek voor algoritmen gebruiken dan bedoelen wij meestal algoritmen die aanvaardbaar snel zijn maar niet gegarandeerd de beste oplossing vinden. Dergelijke algoritmen worden vaak toegepast op problemen waar het vinden van een optimale oplossing veel te duur zou zijn en een algoritme dat tenminste heel vaak een relatief goede oplossing levert een goed compromis is. Omdat jammer genoeg veel problemen die in de praktijk belangrijk zijn van deze soort zijn, zijn heuristieken voor veel toepassingen belangrijk. Metaheuristieken zullen een onderdeel van DA3 zijn, dit zijn manieren om heuristische algoritmen te ontwikkelen.
Een kenmerk van heuristieken is vaak dat beslissingen *lokaal* genomen worden omdat dat goedkoper is. Jullie kunnen jullie voorstellen dat jullie het hoogste punt van de aarde zoeken – maar alleen een omgeving van 20 meter kunnen zien. Dat is natuurlijk maar een grove beschrijving, maar geeft wel een juist idee...

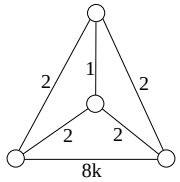
8.1 Gretige algoritmen

De misschien meest gebruikte heuristiek is een greedy of gretige aanzet. Dat betekent dat jullie altijd een beslissing nemen, die op dit moment optimaal lijkt – zoals in het voorbeeld met de bewakers in een museum in het begin van de lesnota's.
Als jullie op een gretige manier een strategie voor het muntspelletje ontwerpen dan zouden jullie bv. in elke zet zoveel mogelijk munten nemen. Dan heb je meer munten als je wint en als je verliest, heeft de andere speler minder. **Lokaal** – dus op dit ene moment is dat dus een goede beslissing. Dat het niet echt een goede strategie is, zie je natuurlijk al als je met 5 munten begint...
De gemakkelijkste manier om het principe (en de problemen) te verstaan is naar (nog) een voorbeeld te kijken:

Stel dat jullie een oplossing voor het handelsreizigersprobleem **moeten** vinden. Dan lijkt een gretige aanzet niet slecht: omdat bogen met een lage kost nuttig lijken voor een goedkope rondreis (tenminste als je geen rekening houdt met globale implicaties) zou je een gretig algoritme als volgt voor het probleem kunnen toepassen:

- begin met een goedkoopste boog in de graaf (als er meerdere zijn, kies om het even welke)
- kies dan een goedkoopste boog van één van de eindpunten naar een nog niet bezochte top tot er geen toppen zijn die nog niet bezocht werden (als er meerdere zijn, kies opnieuw om het even welke)
- kies dan nog de laatste boog tussen de twee eindpunten van het pad

Dat lijkt geen slecht algoritme en misschien zullen in veel gevallen de oplossingen ook niet slecht zijn – maar het zou ook kunnen dat de op deze manier gevonden oplossing heel slecht is. Inderdaad kan de oplossing zelfs een arbitraire factor slechter zijn dan de beste oplossing – en dat al voor kleine grafen:



Figuur 32: Een voorbeeld waar een gretig algoritme voor het handelsreizigersprobleem een heel slechte oplossing vindt. Daarbij staat k voor een arbitrair groot getal.

In Figuur 32 zien jullie een voorbeeld van een graaf van het handelsreizigersprobleem. De beslissing van het gretige algoritme de goedkoopste boog te nemen lijkt lokaal gezien (dus op het eerste gezicht) een goede keuze – maar later ben je dan gedwongen de vreselijk dure boog met kost $8k$ te nemen. Je zal dus een oplossing vinden die meer dan $8k$ kost in plaats van de oplossing met kost 8. Deze oplossing is dus k keer duurder dan de optimale oplossing – om het even hoe groot je k kiest.
Gretige algoritmen kunnen dus soms arbitrair slechte oplossingen geven – maar dat betekent niet dat een gretige aanzet altijd slecht is. Soms is het zelfs zo, dat je van een gretige aanzet kan bewijzen dat die optimale resultaten oplevert.

Een voorbeeld: een buis opdelen

De bedoeling van dit voorbeeld is niet om een optimaal algoritme voor een zeker probleem te geven, maar om te zien, hoe je soms ook voor suboptimale algoritmen prestatiegaranties kan bewijzen en hoe dergelijke garanties soms ook een verband hebben met de parameters.

Als voorbeeld kijken wij naar een probleem dat gelijkaardig is aan Oefening 71: Een buis van lengte l mm moet in kleine stukken gesneden worden. De lengten $l_1, \ldots, l_k \leq l$ kunnen doorverkocht worden – andere lengten zijn niet bruikbaar. De lengten $l_1, \ldots, l_k$ (in mm) en l zijn allemaal gehele getallen. Natuurlijk eist ook hier elke doorsnede een beetje ruimte (1 mm) – dat is dus een deel van de buis dat verloren gaat – net als een mogelijke rest waaruit geen van de stukken meer gesneden kan worden.
Het doel is nu een opdeling te vinden zodat zo weinig mogelijk verloren gaat – dus waar de som van de sneden en de rest zo klein mogelijk is. Wij zullen een gretig algoritme gebruiken. Lokaal lijkt het zinvol altijd een zo groot mogelijke lengte $l_i \in \{l_1, \ldots, l_k\}$ als volgende lengte die gesneden moet worden te kiezen. Natuurlijk mag die ten hoogste de lengte van de rest hebben. Zo heb je grote stukken zonder verlies, terwijl je met meer kleinere stukken ook meer verlies zou hebben. Of als pseudocode:

```
verdeel_buis(l)
// geeft als resultaat het verlies terug: de lengte van de
// sneden plus de lengte van de rest van de buis
{
verlies=0;
rest=l;
l_min= min{l_1,...,l_k};

while (l_min<= rest)
{ // zoek het grootste stuk dat nog past
l_best= max{ l_i | 1<i<=k en l_i <= rest};
rest = rest-l_best-1;
verlies=verlies+1;
}

return verlies+rest;
// klopt ook als rest= -1
}
```

Oefening 86 *Voor gegeven lengten $l_1, ..., l_k$ en variabele lengte l is de complexiteit van dit algoritme $O(l)$. Het is zo geschreven om te beklemtonen dat je elke keer opnieuw de grootst mogelijke lengte kiest.*
Kan je dit gretig algoritme voor gegeven lengten $l_1, ..., l_k$ en variabele lengte l ook in constante tijdt (dus $O(1)$) implementeren?

Oefening 87 *Geef voor dit probleem een algoritme met dynamisch programmeren.*

Dit lijkt een verstandig algoritme omdat je zo weinig mogelijk sneden wilt hebben, dus zo groot mogelijke delen. Je kan ook bewijzen dat het algoritme – ten minste als functie van de lengte van de buis – niet arbitrair slecht kan presteren:

Opmerking 21
- *Als de lengte van de buis l is, zal het gretige algoritme altijd een verlies van ten hoogste $\frac{l}{2}$ hebben.*
- **Maar:** *Er is geen getal k zodat als het minimale verlies m is het gretige algoritme een verlies van ten hoogste $k * m$ berekent.*

Bewijs: Als er een nuttige buislengte van ten minste $\frac{l}{2}$ is, volgt het resultaat onmiddelijk – al naar de eerste stap van het gretige algoritme staat vast dat het verlies ten hoogste $\frac{l}{2}$ is. Stel dus dat de minimale lengte $l_{\min}$ ten hoogste $\frac{l}{2} - 1$ is.

Een bovengrens voor het verlies wordt gegeven door $l_{\min} - 1 + \frac{l}{l_{\min}+1}$. Daarbij wordt gebruikt dat als de laatste rest een gebruikte buislengte is, daar geen snede nodig is. Als je dit naar l_{min} afleidt (resultaat: $1 - \frac{l}{(l_{\min}+1)^2}$) zie je dat er een extremum in het interval zit en als je nog eens afleidt (resultaat: $2\frac{l}{(l_{\min}+1)^3}$) zie je dat het een minimum is. De maximale waarden zitten dus op de rand en daar krijg je voor $l_{\min} = 1$ een verlies van $\frac{l}{2}$ en voor $l_{\min} = \frac{l}{2} - 1$ ook een bovengrens van $\frac{l}{2}$. Inderdaad zou het echte verlies in het tweede geval zelfs kleiner zijn dan de bovengrens. Hiermee volgt het eerste deel.

Voor het tweede deel kijk naar het geval waar je voor een arbitraire oneven lengte l twee lengten $l_1 = \frac{l-1}{2}$ en $l_2 = \frac{l+1}{2}$ hebt. Het gretige algoritme zal eerst l_2 kiesen en dus een verlies van $\frac{l-1}{2}$ hebben, terwijl duidelijk is dat een verlies van 1 mogelijk is door twee keer l_1 te kiesen.

■

Het lijkt dus alsof ook hier een gretig algoritme slecht presteert. Maar het zou best kunnen dat de nuttige buislengten altijd dezelfden zijn en alleen de lengten van de hele buis verschillen. Voor sommige van de buislengten kan je dan goede prestatiegaranties bewijzen en soms zelfs dat het gretige algoritme het optimum vindt (bv. voor $l_1 = 4, l_2 = 9, l_3 = 14$) en veel andere combinaties.

Opmerking 22 *Als de maximale toegelaten buislengte $l_{\max} \geq 2$ is en een optimale oplossing heeft een verlies van m, dan vindt het gretige algoritme een oplossing met verlies ten hoogste $m + l_{\max} - 2$.*

Bewijs: Stel dat de door een optimaal algoritme gekozen lengten in dalende volgorde $o_1, \ldots, o_k$ zijn en dat de door het gretige algoritme gekozen lengten $g_1, \ldots, g_n$ zijn. Daarbij duiken lengten die meerdere keren gebruikt worden ook meerdere keren op. Dus zijn k, resp. n de aantallen buizen. Stel dat j de grootste index is zodat $g_j = l_{\max}$. Dan is $k \geq j$ omdat anders de optimale oplossing beter gemaakt zou kunnen worden door nog een buis van lengte – bv. – $l_{\max}$ te snijden en als $k = j$ dan is het resultaat van het gretige algoritme optimaal. Stel dus dat $k > j$ en dus $m \geq k - 1 \geq j$. Aan de andere kant is het verlies dat het gretige algoritme berekent ten hoogste $j + l_{\max} - 2 \leq m + l_{\max} - 2$ omdat de rest ten hoogste $l_{\max} - 1$ is en als de rest $l_{\max} - 1$ is en er wordt geen andere buislengte meer gekozen, dan is $l_{\max}$ de enige lengte en de oplossing dus optimaal.

Dat deze grens in feite bereikt kan worden, hebben wij al gezien.

■

Als de toegelaten buislengten dus allemaal relatief klein zijn vergeleken met de totale lengte van de buis, dan is de door een gretig algoritme berekende oplossing dus mischien goed genoeg!

Oefening 88

- *Bewijs expliciet dat het gretige algoritme voor het opdelen van buizen voor toegelaten buislengten van $l_1 = 4, l_2 = 9, l_3 = 14$ optimaal presteert.*

- *Geef het maximale verschil tussen het minimale verlies en het verlies dat door het gretige algoritme wordt berekend als functie van de minimale nuttige buislengte als dat kan. Anders bewijs dat het niet kan.*

Oefening 89 *Geef een algoritme met dynamisch programmeren dat de het probleem een opdeling met minimaal verlies te berekenen even snel (volgend de $O()$-notatie) oplost als het gretige algoritme maar in tegenstelling tot de heuristiek natuurlijk altijd een optimale oplossing vindt.*

Oefening 90 *Een* matching *of* koppeling *in een graaf $G = (V, E)$ is een deelverzameling M van de boogverzameling E, zodat geen twee bogen in M een gemeenschappelijke top hebben.*

- *Geef een gretig algoritme dat probeert een maximale koppeling te construeren en gegarandeerd een koppeling vindt met grootte $c * m(G)$ waarbij $c > 0$ een constante is en $m(G)$ de maximale grootte van een koppeling in G. Geef expliciet de eigenschap die het volgens jou tot een gretig algoritme maakt.*

- *Bewijs dat het algoritme altijd een koppeling van grootte $c * m(G)$ vindt (voor een constante $c > 0$) en geef de waarde van c voor jouw algoritme.*

Een voorbeeld: het Yutsis probleem

Ten slotte zullen wij nog een voorbeeld zien dat met het onderzoek aan deze universiteit te maken heeft. Het is uit *D. Van Dyck, G. Brinkmann, V. Fack en B.D. McKay. To be or not to be Yutsis: algorithms for the decision problem, Computer Physics Communications 173, 1–2: 61–70 (2005).* Het geeft jullie ook een voorbeeld van het soort onderzoek dat jullie in jullie masterthesis zouden kunnen doen als jullie later de master informatica volgen en voor een algoritmisch georiënteerde thesis kiezen.
Omdat het hier om een beslissingsprobleem gaat (het antwoord is dus *ja* of *nee*), en het antwoord niet in alle gevallen juist is, kan je hier natuurlijk niets (zinvols) daarover bewijzen hoe good het juiste resultaat benaderd wordt. Het is toch al een voorbeeld dat toont hoe krachtig gretige algoritmen soms kunnen zijn.

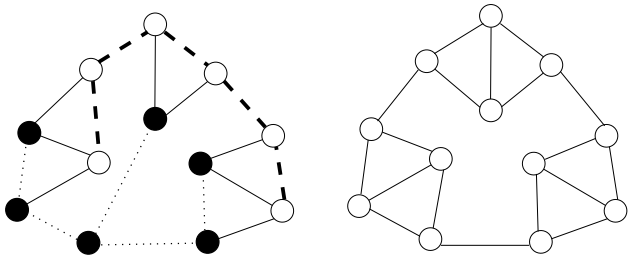
Definitie 18 *Gegeven een graaf $G = (V, E)$ met $2 * n$ toppen voor een zekere n waarin elke top graad 3 heeft – dat noemen wij een 3-reguliere graaf.*
Als $V' \subseteq V$ dan noemen wij de graaf (V', E') waarbij E' de verzameling van alle bogen uit E is die beide eindpunten in V' hebben de door V' geïnduceerde deelgraaf.

Een opdeling van de toppenverzameling in 2 disjuncte delen D_1, D_2 heet een Yutsis-decompositie als zij de volgende eigenschappen heeft:

- *elk deel D_i heeft n toppen*
- *voor elk deel D_i is de door D_i geïnduceerde deelgraaf een boom.*

Een Yutsis-graaf is een graaf die een Yutsis-decompositie heeft.

Yutsis-decomposities zijn belangrijk in de theoretische fysica, maar daar zullen wij het hier niet over hebben.
Een voorbeeld van een graaf met een Yutsis-decompositie en van een graaf waarvoor geen Yutsis-decompositie bestaat toont Figuur 33.
Het doel is nu voor een gegeven 3-reguliere graaf een Yutsis-decompositie te vinden – als die een Yutsis-decompositie heeft. Jammer genoeg is het al NP-compleet alleen maar te beslissen of een gegeven graaf een Yutsis-decompositie heeft. Voor de deelklasse van grafen die je in het vlak kan tekenen zonder dat bogen kruisen bestaan soms efficiëntere algoritmen dan voor algemene grafen, maar zelfs voor deze deelklasse blijft het probleem



Figuur 33: Een voorbeeld van een graaf met een Yutsis-decompositie en van een graaf waarvoor geen Yutsis-decompositie bestaat.

NP-compleet. Het is dus zeker zinvol een heuristiek te gebruiken, die ten minste in veel gevallen relatief snel een Yutsis-decompositie vindt als die er is.
Wij werken als volgt: Als de gegeven 3-reguliere graaf $2n$ toppen heeft, proberen wij een verzameling M van n toppen te vinden zodat de door M geïnduceerde deelgraaf een boom is en de door het complement $V \setminus M$ geïnduceerde deelgraaf samenhangend. Je kan gemakkelijk aantonen dat die dan inderdaad ook een boom is (zie Opmerking 23).
Wij beginnen met stap 1 en een verzameling $M_1 = \{v_1\}$ met een toevallig gekozen top v_1. In elke stap i hebben we een verzameling M_i die een boom induceert en een verzameling L_i met de volgende eigenschappen:

(i) elke top $v \in L_i$ heeft precies één buur in M_i

(ii) voor elke $v \in L_i$ is de door $V \setminus (M_i \cup \{v\})$ geïnduceerde graaf samenhangend.

Dat betekent dat je elke top $v \in L_i$ aan M_i kan toevoegen en $M_{i+1} = M_i \cup \{v\}$ induceert een boom met een samenhangend complement.
L_1 kan je gemakkelijk opbouwen door eigenschap (i) en (ii) expliciet te toetsen. Het kan in lineaire tijd getoetst worden, welke toppen aan eigenschap (ii) voldoen (zie les *algoritmische grafentheorie*). Als je een nieuwe top aan een M_i toevoegt, worden zijn buren die nog niet in $M_i \cup L_i$ zijn aan L_i toegevoegd en achteraf wordt voor elke top in de nieuwe verzameling L_{i+1} getest of hij aan (i) en (ii) voldoet. Als dat niet het geval is, wordt hij verwijderd. Eigenschap (i) moet natuurlijk alleen maar voor de buren van de nieuwe top in L_i getoetst worden.
Het algoritme werkt nu heel eenvoudig:

```
vind_Yutsis(G=(V,E))
{
```

```
n=|V|/2
M = {v_1} waarbij v_1 toevallig gekozen is
bereken L

while (L niet leeg)
{ kies top v uit L die de meeste buren buiten L heeft
  M = M + {v}
  update L
}
if (M bevat n elementen) return "gevonden";
else return "niet gevonden"
}
```

Wij willen dat L zo laat mogelijk leeg is. Lokaal lijkt het dus een goed idee te proberen L zo groot mogelijk te maken zodat er – ook al verwijder je toppen – toch nog toppen over zijn voor de volgende stap. De stap `kies top v uit L die de meeste buren buiten L heeft` is dus typisch voor een gretig algoritme: zo heb je zo veel mogelijk nieuwe kandidaten om aan L toe te voegen, het lijkt dus op dit moment een goede beslissing. Merk op dat het aantal buren in M voor elke top in L gelijk is: één .

Opmerking 23 *Als het algoritme opgestart op een 3-reguliere graaf $G = (V, E)$ met $2n$ toppen een verzameling M_n kan bouwen, vormen M_n en $V \setminus M_n$ een Yutsis-decompositie.*

Bewijs: Omdat M_n n toppen bevat, is de som van de graden in G gelijk aan $3 * n$. Een boom met n toppen heeft $n - 1$ bogen, dus behoren $2(n - 1)$ van deze graden tot bogen waarvan beide eindpunten in M zijn en $3n - 2(n - 1) = n + 2$ bogen hebben één eindpunt in M en één eindpunt in $V \setminus M_n$. Dus heeft $V \setminus M_n$ precies $3n - n + 2 = 2(n - 1)$ graden die tot bogen behoren met beide eindpunten in $V \setminus M_n$. Er zijn dus $n-1$ bogen tussen toppen in $V \setminus M_n$ en omdat $V \setminus M_n$ samenhangend is, is de door $V \setminus M_n$ geïnduceerde graaf dus ook een boom. ■

Oefening 91 *Bewijs dat het algoritme opgestart op een graaf met $2n$ toppen nooit een verzameling M_{n+1} maakt.*

Dus geeft het algoritme `"gevonden"` terug als en slechts als het een Yutsis-decompositie heeft gevonden en gaat nooit verder dan n en geeft `"niet gevonden"` terug hoewel het tijdens de constructie al een decompositie had gevonden.
Hier heb je geen verhouding tussen de gevonden oplossing en de beste oplossing – alle Yutsis-decomposities zijn even goed – de interessante vraag is

dus: in hoeveel gevallen vindt deze eenvoudige gretige heuristiek een decompositie als die er is? In feite zijn er gevallen waar de heuristiek – om het even met welke top je begint en welke van de *beste* toppen je kiest – nooit een decompositie zal vinden. De kleinste dergelijke graaf heeft 22 toppen. Maar dergelijke grafen zijn **extreem** uitzonderlijk. De heuristiek werd voor 350.000 toevallig gekozen 3-reguliere grafen met t.e.m. 300.000 toppen toegepast. Daarbij werden 25.000 grafen met 300.000 toppen getest. Op één graaf na werd er voor elke graaf een decompositie gevonden. Gemiddeld waren er minder dan 1.2 pogingen nodig en maximaal 8 pogingen. De ene graaf waar er geen decompositie werd gevonden bleek achteraf een brug te hebben – die had dus gewoon geen decompositie.
Dit is dus een heel moeilijk – NP-compleet – probleem dat een heel eenvoudige gretige heuristiek in bijna alle gevallen heel snel kan oplossen.
De conclusie is dus aan de ene kant dat NP-complete problemen niet altijd hopeloos zijn (ten minste gemiddeld), maar ook dat gretige heuristieken soms heel krachtig kunnen zijn!

Oefening 92 *Je hebt bestanden $B_1, \ldots, B_n$ gevuld met een verschillend aantal getallen die al gesorteerd zijn en die tot één groot bestand gemerged moeten worden. Daarbij bevat voor $1 \leq i \leq n$ bestand B_i precies l_i getallen. In elke stap worden 2 bestanden gemerged en de kost om een bestand met l_i getallen en een bestand met l_j getallen te mergen is $l_i + l_j$.*
Wij zoeken nu de optimale volgorde om de bestanden te mergen.

a.) *Geef een voorbeeld dat aantoont dat verschillende volgordes tot verschillende kosten kunnen leiden.*

b.) *Toon expliciet aan dat altijd $n - 1$ mergeoperaties nodig zijn.*

c.) *Beschrijf een gretig algoritme dat probeert een volgorde met minimale kost te bepalen. Denk je dat jouw algoritme misschien zelfs een optimale oplossing vindt?*

8.2 Online algoritmen

Wij zullen een gretige aanzet ook gebruiken voor een probleem voor online algoritmen. Inderdaad hebben jullie al online(-achtige) algoritmen gezien zonder die zo te noemen. Online algoritmen moeten een taak oplossen hoewel ze nog niet alle data hebben – als nieuwe data komt, moet onmiddellijk een oplossing voor de tot nu toe gekregen data gevonden worden zonder ermee rekening te kunnen houden welke data nog komt. De nieuwe oplossing is daarbij gebaseerd op de oude oplossing. Hier zijn dus gretige algoritmen

die alleen een lokaal goede oplossing zoeken bijna typisch – ten slotte heb je de globale informatie zelfs niet...
Stel bv. dat het probleem is voor een gegeven verzameling van sleutels een binaire hoop op te bouwen die deze sleutels bevat. In DA1 hebben jullie gezien dat dat in lineaire tijd kan gebeuren. Maar dat is normaal niet de manier waarop je een hoop gebruikt: je weet normaal niet op voorhand welke sleutels er toegevoegd moeten worden maar je krijgt de sleutels gewoon na elkaar om ze toe te voegen en na elke sleutel moet je ervoor zorgen dat de datastructuur een binaire hoop is – om het even of daarna nog sleutels toegevoegd moeten woden of niet! Ook voor binaire hopen moet je voor de eigenschap *online* te kunnen werken een prijs betalen: het toevoegen van n sleutels heeft dan een kost van $n \log n$ in plaats van n.
In het geval van binaire zoekbomen met n sleutels is de kost voor het (online) bouwen van een bv. rood-zwart boom even groot als voor het (offline) opbouwen van een boom – in beide gevallen $O(n \log n)$. Maar offline kan je met deze kost zelfs een **optimale** boom opbouwen – dus met diepte $\log n$ in plaats van $2 \log(n + 1)$.
Er zijn natuurlijk ook problemen die online even snel opgelost kunnen worden als offline (bv. het maximum in een verzameling zoeken), maar in de meeste gevallen presteren online algoritmen toch slechter: ofwel vragen ze meer tijd ofwel zijn de resultaten minder goed.
Online algoritmen zijn belangrijk, omdat in de praktijk vaak online problemen voorkomen: Een typische toepassing is bv. als je dingen wil opslaan in een pakhuis – dan weet je ook niet altijd op voorhand wat er opgeslaan moet worden. Of het zou ook kunnen dat een handelsreiziger als hij vertrekt nog niet alle steden kent die hij moet bezoeken en zijn toer dan *online* opgesteld moet worden. Bewerkingen op databanken, zoekbomen, en heaps zijn allemaal online, etc.

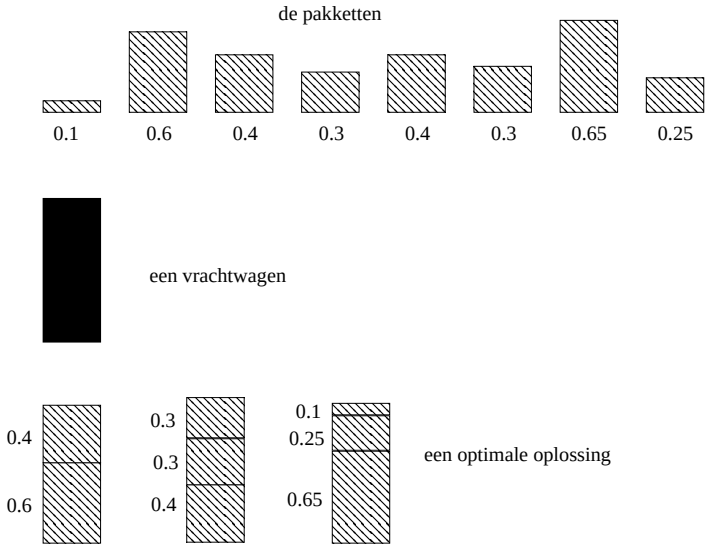
In het volgende zullen wij nog meer eisen van online algoritmen. Wij willen niet alleen dat de nieuwe oplossing *gebaseerd* is op de oude, maar dat de oude oplossing een ongewijzigde deeloplossing van de nieuwe is. Bij zoekbomen hebben wij wel een aantal wijzigingen toegelaten – ook al was het aantal wijzigingen klein in vergelijking met de grootte van de hele boom. Als je voor zoekbomen eist dat er helemaal geen wijzigingen gebeuren, heb je een ongebalanceerde zoekboom – en die presteert natuurlijk heel slecht...
Uitleg (geen definitie):
Gegeven een rij van inputs $i_1, \ldots, i_n$ en een probleem dat voor dergelijke rijen van inputs gedefinieerd is.
Een algoritme dat eerst het probleem voor i_1, dan voor i_1, i_2, dan i_1, i_2, i_3 etc. oplost en de oplossing voor $i_1, i_2, \ldots, i_{n-1}$ als deeloplossing voor de oplossing

van $i_1, i_2, \ldots, i_n$ gebruikt noemen wij een online algoritme.

Om daarvan een echte definitie te maken zouden wij natuurlijk expliciet moeten zeggen wat wij bedoelen met *als deeloplossing gebruikt*.

8.2.1 Het online inpakprobleem

Wij willen het volgende probleem online oplossen waarvoor jullie in de oefeningen al een branch and bound algoritme hebben ontworpen:
Het inpakprobleem:
Gegeven een rij $g_1, \ldots, g_n$ van gewichten van pakketten $p_1, \ldots, p_n$ die vervoerd moeten worden. Je hebt identieke vrachtwagens waarop je de pakketten kan plaatsen maar je moet er wel op letten dat de capaciteit C van de vrachtwagens niet overschreden wordt. Het doel is natuurlijk de pakketten op een manier op de vrachtwagens te plaatsen dat zo weinig mogelijk vrachtwagens gebruikt worden.
Natuurlijk kunnen wij het probleem herschalen (en dat zullen wij doen). Wij stellen dus dat $C = 1$ en dat voor $1 \leq i \leq n$ geldt $g_i \leq 1$ (anders kunnen de pakketten niet geplaatst worden).



Figuur 34: Een voorbeeld voor het inpakprobleem.

Hier betekent *deeloplossing gebruiken* dat zodra een pakket geplaatst werd het niet meer naar een andere vrachtwagen verplaatst mag worden. Wij plaatsen dus eerst pakket p_1 (zonder de andere gewichten te kennen), dan pakket p_2 waarbij wij de oplossing voor p_1 kennen maar niet mogen wijzigen, etc. Als pakket p_i geplaatst moet worden, zijn $p_1, \ldots, p_{i-1}$ al geplaatst en

mogen niet meer verplaatst worden. De gewichten van de volgende pakketten zijn nog niet gekend (wij weten zelfs niet **of** er volgende pakketten zijn).

Oefening 93 *Stel dat je het inpak-beslissingsprobleem in polynomiale tijd kan oplossen. Dus dat er een polynomiaal algoritme bestaat dat voor een gegeven rij $g_1, \ldots, g_k$ van gewichten en een gegeven getal n kan bepalen of n vrachtwagens voldoende zijn om de pakketten te vervoeren. De tijd is dan een polynoom in k.*
Toon aan:

- *Dan bestaat er ook een polynomiaal algoritme dat het minimale aantal vrachtwagens voor een gegeven rij $g_1, \ldots, g_k$ kan bepalen.*
- *Dan bestaat er ook een polynomiaal algoritme dat voor een gegeven rij $g_1, \ldots, g_k$ en een gegeven rij $c_1, \ldots, c_m$ van capaciteiten van m vrachtwagens (die dus* **niet** *gelijk moeten zijn) kan bepalen of deze m vrachtwagens voldoende zijn om de gewichten te vervoeren.*
- *Dan bestaat er ook een polynomiaal algoritme dat voor een gegeven rij $g_1, \ldots, g_k$ en een gegeven rij $c_1, \ldots, c_m$ van capaciteiten van m vrachtwagens een optimale plaatsing kan bepalen.*

De belangrijkste vraag is natuurlijk of je nog altijd de optimale oplossing kan vinden...

Oefening 94 *Toon aan: Voor elke rij $g_1, \ldots, g_n$ van gewichten bestaat er een online algoritme dat de optimale oplossing voor de hele rij vindt.*

Maar zoals altijd in stellingen is het **heel** belangrijk waar de kwantoren staan:

Stelling 24 *Er bestaat geen online algoritme dat voor alle inputrijen de optimale oplossing vindt.*

Of iets preciezer: Voor elk online inpakalgoritme en elke $k \in \mathbb{N}$ bestaat er een inputrij zodat het algoritme ten minste $\frac{4}{3}m$ vrachtwagens gebruikt waarbij m het aantal vrachtwagens in een optimale oplossing is en $m \geq k$.

Bewijs: Het idee van dit bewijs is dat wij naar 2 rijen kijken waarvan de ene een deelrij van de andere is. Stel dus dat k en een arbitrair algoritme A gegeven zijn.

De ene rij $R_1 = g_1, \ldots, g_n$ waarnaar wij kijken, heeft $n = 4k$ gewichten waarbij de gewichten $g_1, \ldots g_{2k}$ allemaal $\frac{1}{2} - \epsilon$ zijn voor een heel kleine (maar positieve) epsilon en de gewichten $g_{2k+1}, \ldots, g_{4k}$ zijn allemaal

$\frac{1}{2}+\epsilon$. De andere rij R_2 waarnaar wij kijken, is de deelrij bestaande uit de eerste $2k$ gewichten $g_1, \ldots g_{2k}$.

Het optimum voor R_2 is duidelijk als je twee gewichten op elke vrachtwagen pakt – dus heb je k vrachtwagens nodig. Als het algoritme ten minste $\frac{4}{3}k$ gebruikt, hebben wij onze rij gevonden.

Nu zullen wij bewijzen dat als het algoritme minder dan $\frac{4}{3}k$ vrachtwagens gebruikt en wij de gewichten niet mogen herverdelen (omdat het online is) het algoritme voor de hele rij ten minste $\frac{4}{3}$ keer het optimum gebruikt – gewoon omdat het niet beter **kan** als je de verdeling van de eerste gewichten niet mag wijzigen!

Op elke vrachtwagen kunnen ten hoogste 2 gewichten geplaatst worden. Stel dat er a_1 pakketten alleen op een vrachtwagen zitten en a_2 samen met een ander pakket. Dan is $a_1 + a_2 = 2k$ dus (a)$a_1 = 2k - a2$ en (b)$\frac{a_2}{2} + a_1 < \frac{4}{3}k$. Als wij nu (a) in (b) invoegen krijgen wij $\frac{a_2}{2} > \frac{2}{3}k$. Maar op de $\frac{a_2}{2}$ vrachtwagens waar er al twee pakketten zitten (dus waar het gewicht al $1-2\epsilon$ is) kan je geen van de overblijvende pakketten van rij R_1 plaatsen – en je kan ook geen twee van de laatste $2k$ gewichten op dezelfde vrachtwagen plaatsen omdat de som van de gewichten van 2 pakketten groter dan 1 is. Voor de hele rij R_1 moet het algoritme (omdat het de eerste $2k$ gewichten niet mag verplaatsen) dus meer dan $\frac{2}{3}k + 2k = \frac{8}{3}k$ vrachtwagens gebruiken. Maar het optimum is duidelijk $2k$ (als je op elke vrachtwagen één pakket met gewicht $\frac{1}{2} - \epsilon$ en één pakket met gewicht $\frac{1}{2} + \epsilon$ plaatst). Het algoritme gebruikt dus meer dan $\frac{8}{3}/2 = \frac{4}{3}$ keer het optimale aantal vrachtwagens $2k$.

■

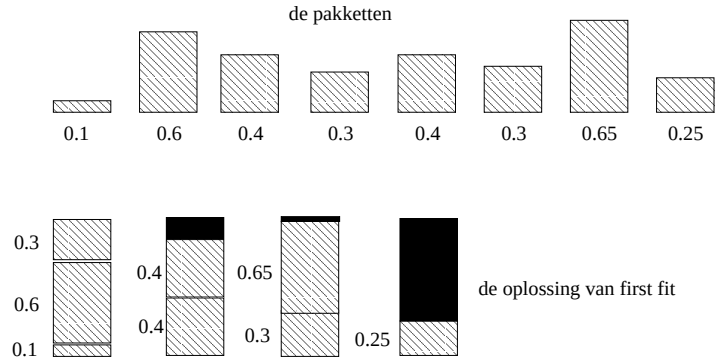
In dit bewijs is ϵ geen parameter – het is gewoon een getal. Het is gewoon gemakkelijker te verstaan als je schrijft $\frac{1}{2} + \epsilon$ in plaats van 0.50037 of zoiets omdat zo het idee *net iets groter dan* $\frac{1}{2}$ duidelijker wordt. Maar wij moeten ons er natuurlijk van overtuigen dat er een getal voor ϵ bestaat dat ervoor zorgt dat alle argumenten werken! Wij hebben nodig dat de gewichten positief zijn, dus $\frac{1}{2} - \epsilon > 0 \Rightarrow \epsilon < \frac{1}{2}$. Wij hebben ook nodig dat je geen twee pakketten met gewicht $\frac{1}{2} + \epsilon$ op één vrachtwagen kan plaatsen – dus $2 * (\frac{1}{2} + \epsilon) > 1 \Rightarrow \epsilon > 0$. Ten slotte hebben wij nog nodig dat je geen 3 pakketten met gewicht $\frac{1}{2} - \epsilon$ op één vrachtwagen kan plaatsen – dus $3 * (\frac{1}{2} - \epsilon) > 1 \Rightarrow \epsilon < \frac{1}{6}$ en dat je geen twee pakketten met gewicht $\frac{1}{2} - \epsilon$ en één pakket met gewicht $\frac{1}{2} + \epsilon$ samen kan plaatsen – dus $3 * \frac{1}{2} - \epsilon > 1 \Rightarrow \epsilon < \frac{1}{2}$. Wij kunnen dus om het even welk getal ϵ kiezen dat aan $0 < \epsilon < \frac{1}{6}$ voldoet – bv. 0.00037.

Oefening 95 *Bewijs de volgende stelling:*
Voor elk online inpakalgoritme bestaat er een inputrij zodat het algoritme ten minste $\frac{3}{2}m$ vrachtwagens gebruikt waarbij m het aantal vrachtwagens in een optimale oplossing is.

Wij weten dus al op voorhand dat om het even hoe slim ons online algoritme ontworpen is – in sommige gevallen kan het gewoon geen oplossing vinden die even goed is als een algoritme dat offline kan werken. Nu zullen wij sommige voorbeelden van (gemakkelijke) online algoritmen zien.

First fit strategie:
De first fit strategie (of first fit heuristiek) doet precies wat de naam ons doet verwachten:

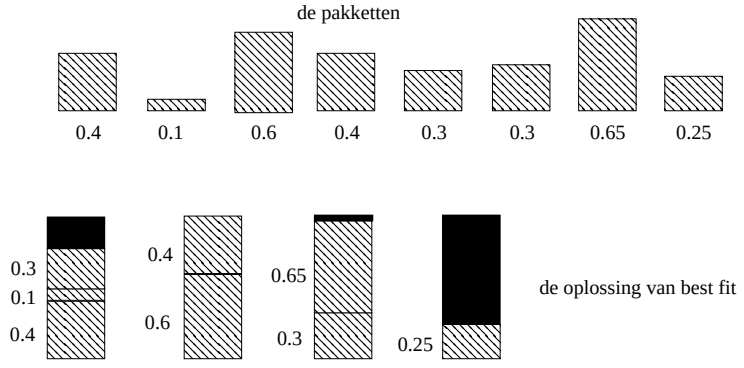
- Als pakketten $p_1, \ldots, p_i$ al geplaatst zijn (waarbij i ook gelijk aan 0 kan zijn) (bv. op vrachtwagens $v_1, \ldots, v_k$) en pakket p_{i+1} moet geplaatst worden, dan plaats het pakket op de vrachtwagen met de kleinste index waarop nog voldoende ruimte is als er zo'n vrachtwagen bestaat en anders op een nieuwe vrachtwagen met index $k + 1$.



Figuur 35: Een voorbeeld voor first fit toegepast op het voorbeeld van Figuur 34.

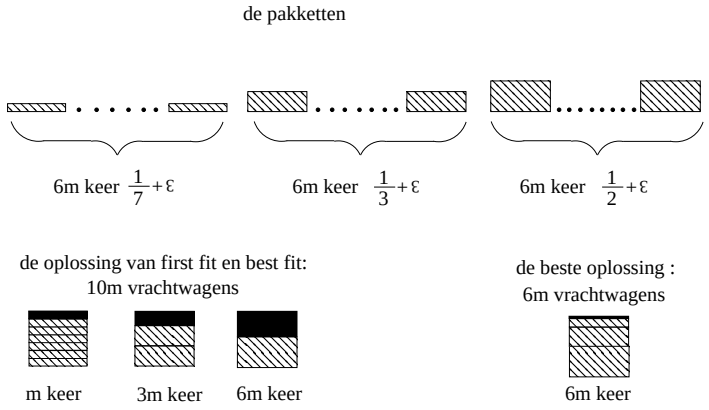
Maar het lijkt niet echt slim er niet meer rekening mee te houden hoeveel vrije ruimte nog op de vrachtwagens is dan gewoon of een gewicht er nog geplaatst kan worden of er niet meer geplaatst kan worden. Misschien lijkt het volgende algoritme slimmer:
Best fit strategie:
Ook de best fit strategie (of best fit heuristiek) doet precies wat de naam ons doet verwachten:

- Als pakketten $p_1, \ldots, p_i$ al geplaatst zijn (waarbij i ook gelijk aan 0 kan zijn) (bv. op vrachtwagens $v_1, \ldots, v_k$) en pakket p_{i+1} moet geplaatst worden, plaats het pakket dan op een vrachtwagen waar er nog voldoende ruimte is en waar de vrije ruimte zo dicht mogelijk bij het gewicht van het pakket ligt. Als zo'n vrachtwagen niet bestaat, plaats het op een nieuwe vrachtwagen met index $k + 1$.



Figuur 36: Een voorbeeld voor best fit toegepast op de gewichten van Figuur 34 maar in een andere volgorde.

Het is gemakkelijk voor elke m een rij van gewichten aan te geven waarvoor first fit en best fit een oplossing berekenen die duidelijk slechter is dan het optimum van m vrachtwagens. In Figuur 37 zien jullie een voorbeeld waar first fit en best fit 5/3 keer het optimale aantal vrachtwagens gebruiken.



Figuur 37: Een rij waarvoor first fit en best fit 5/3 keer het optimale aantal vrachtwagens gebruiken.

Oefening 96 *Welke getallen kan je voor de ϵ in Figuur 37 gebruiken zodat de argumenten die nodig zijn om te zien dat van first fit en best fit $5/3$ keer het optimale aantal vrachtwagens gebruikt wordt allemaal werken?*

Maar inderdaad is dit gemakkelijke voorbeeld al bijna het slechtste geval voor first fit en best fit:

Stelling 25 *Als een rij van pakketten optimaal op m vrachtwagens geplaats kan worden dan gebruiken first fit en best fit ten hoogste $\lceil \frac{17}{10}m \rceil$ vrachtwagens. Dit is een goede bovengrens omdat voor beide algoritmen rijen bestaan zodat ze $\lceil \frac{17}{10}(m-1) \rceil$ vrachtwagens gebruiken.*

Een bewijs vinden jullie bv. in *Worst-case performance bounds for simple one-dimensional packing problems* van Johnson, Demers, Ullman Garey and Graham (1974). Maar omdat het bewijs geen leuke ook in een bredere context bruikbare ideeën geeft en ook vrij ingewikkeld is en meerdere gevallen moet beschouwen, is het niet nuttig het hier te geven.
Ervaring: in de praktijk presteren de algoritmen duidelijk beter.

Oefening 97 *Definieer een algoritme voor het inpakprobleem als* zwak online *als voor een rij $g_1, \ldots, g_n$ van gewichten elk gewicht g_i onmiddellijk geplaatst wordt – zonder rekening te houden met de gewichten $g_{i+1}, \ldots, g_n$ (dat is zoals met online algoritmen) maar elke keer dat een nieuw gewicht g_i wordt geplaatst, mag één van de gewichten $g_1, \ldots, g_{i-1}$ op een nieuwe plaats gezet worden (dat mag een online algoritme niet).*
Schrijf voor elk van de twee volgende zinnen of ze juist zijn en toon aan dat jouw antwoord juist is:

a.) *Er bestaat een oneindige rij van inputsets $i_1, \ldots, i_m$ voor het inpakprobleem zodanig dat voor elke i_k in de rij ten minste k vrachtwagens nodig zijn en elk zwak online algoritme voor elke input i_k uit deze rij ten minste $4/3$ keer het optimum aantal vrachtwagens gebruikt.*

b.) *Voor elk zwak online algoritme A bestaat een oneindige rij van inputsets $i_1, \ldots, i_m$ zodanig dat voor elke i_k in de rij ten minste k vrachtwagens nodig zijn en A voor elke input uit deze rij ten minste $4/3$ keer het optimum vrachtwagens gebruikt.*

Tip: Denk aan de rij uit de les, maar splits de kleine stukken in zoveel nog kleinere stukken op dat herverdelen niet meer mogelijk is als de grote stukken worden geplaatst.

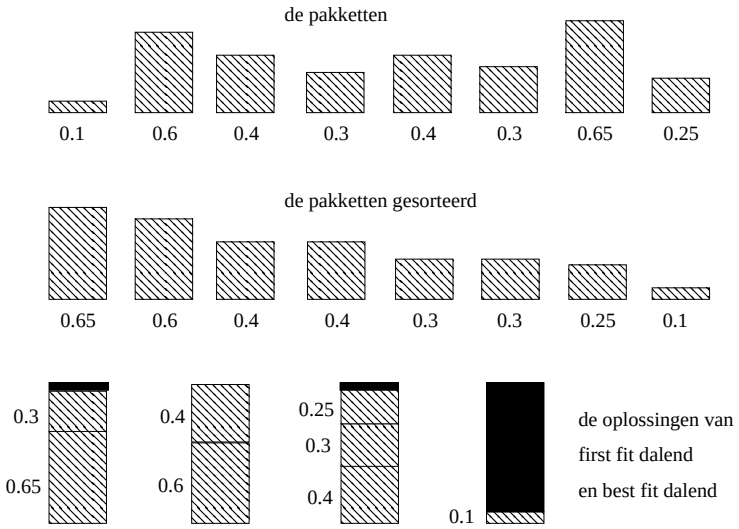
8.2.2 Offline inpakheuristieken

Wij kunnen onze online algoritmen ook gebruiken om heuristieken te vinden voor het offline inpakprobleem.
Een motivatie zou de volgende oefening kunnen zijn:

Oefening 98 *Toon aan:*
Voor elke rij $g_1, \ldots, g_n$ van gewichten bestaat er een volgorde $g_{i_1}, \ldots, g_{i_n}$ zodat first fit **en** *best fit toegepast op de rij in de nieuwe volgorde een optimale oplossing vinden.*

Bestaat er voor elke optimale oplossing ook een volgorde zodat first fit en best fit de gewichten zo plaatsen dat dezelfde gewichten op een vrachtwagen samen zitten?

In plaats van een heuristiek te bedenken om de gewichten goed te plaatsen zouden wij dus ook over een heuristiek om een goede volgorde te vinden kunnen nadenken. Het probleem met de rijen in Stelling 24 en Figuur 37 was dat de grote gewichten eerst laat kwamen en je de kleine gewichten dan niet meer had om ze *in gaten te stoppen*. Het lijkt duidelijk een betere volgorde als je de gewichten in dalende volgorde sorteert.
De (offline) heuristiek die de gewichten eerst in dalende volgorde sorteert en dan first fit (resp. best fit) toepast heet first fit dalend (resp. best fit dalend).



Figuur 38: First fit dalend en best fit dalend vinden dezelfde (niet optimale) oplossing voor de gewichten van Figuur 34 .

Inderdaad zullen wij zien dat wij voor first fit dalend en best fit dalend een performantiegarantie kunnen bewijzen die duidelijk beter is dan voor de online heuristieken:

Stelling 26 *Als het optimale aantal vrachtwagens voor een gegeven rij van gewichten m is, dan gebruiken first fit dalend en best fit dalend ten hoogste $\lceil \frac{4}{3}m \rceil$ vrachtwagens.*

Bewijs: In het bewijs zullen wij altijd alleen first fit dalend schrijven maar je kan dat gewoon elke keer vervangen door *„first fit dalend of best fit dalend“* – natuurlijk moet je daarbij goed nadenken...

Dit zal een gevolg zijn van twee opmerkingen:

Opmerking 1: Als een inputrij $g_1, \ldots, g_n$ optimaal op m vrachtwagens geplaatst kan worden dan plaatsen first fit dalend en best fit dalend op vrachtwagens met een index groter dan m alleen maar pakketten met een gewicht van ten hoogste $\frac{1}{3}$.

Bewijs:

Stel dat dit niet het geval is en $g_1, \ldots, g_n$ de inputrij na het sorteren is. Bovendien zij g_i het eerste gewicht dat van first fit dalend op een vrachtwagen met index ten minste $m+1$ geplaatst wordt. Dan is het ook automatisch het grootste dergelijke gewicht en dus geldt $g_i > \frac{1}{3}$. Omdat de gewichten gesorteerd zijn, geldt $g_j > \frac{1}{3} \quad \forall 1 \leq j \leq i$. Dus zitten er op elke vrachtwagen ten hoogste twee gewichten – om het even hoe ze geplaatst worden.

Nu kijken wij naar de verdeling van de eerste i gewichten door het optimale algoritme en door first fit dalend: Stel dat het aantal gewichten dat door het optimale algoritme met k gewichten op een vrachtwagen wordt geplaatst o_k is en analoog f_k voor first fit dalend.

Dan geldt

$$\frac{o_2}{2} + o_1 \leq m \qquad \text{en} \qquad o_2 + o_1 = i$$

dus

$$o_1 = i - o_2$$

en

$$i - \frac{o_2}{2} \leq m$$

Wij hebben ook

$$\frac{f_2}{2} + f_1 = m + 1 \qquad \text{en} \qquad f_2 + f_1 = i$$

dus

$$i - \frac{f_2}{2} - 1 = m$$

en samengevat

$$f_2 \leq o_2 - 2$$

en dus $$f_1 \geq o_1 + 2.$$

Dus zijn er in een optimale plaatsing ten minste twee gewichten die dat met zijn tweeën op een vrachtwagen zitten (waarbij wij alleen naar gewichten met een index $j \leq i$ kijken) terwijl first fit dalend deze gewichten alleen plaatst. Er is ten minste één dergelijk gewicht g_t met een index die niet i is. Maar het gewicht dat samen met g_t geplaatst wordt moet ten minste even groot zijn als g_i – dus zou er ook g_i geplaatst kunnen worden. Maar dan zou first fit dalend geen nieuwe vrachtwagen aangemaakt hebben (een tegenstrijdigheid).

(Einde bewijs opmerking.)

Opmerking 2:

Het aantal objecten dat door first fit dalend en best fit dalend op vrachtwagens met een index $j > m$ wordt geplaatst, is ten hoogste $m - 1$.

Bewijs:
Omdat het mogelijk is de gewichten op m vrachtwagens te plaatsen geldt natuurlijk

$$\sum_{j=1}^{n} g_j \leq m$$

Als w_j de som van de door first fit dalend op vrachtwagen j geplaatste gewichten is en $t_1, \ldots, t_k$ de gewichten zijn die op een vrachtwagen met index ten minste $m + 1$ geplaatst worden, geldt

$$\sum_{j=1}^{m} w_j + \sum_{j=1}^{k} t_j \leq m$$

Maar wij hebben voor elke j zeker dat $w_j + t_j > 1$ omdat gewicht t_j anders op vrachtwagen j geplaatst zou worden. ((*)Voor deze stap moet je over best fit dalend nadenken. . .). Als er dus ten minste m gewichten op een vrachtwagen met index $j > m$ geplaatst zouden worden (dus $k \geq m$):

$$m \geq \sum_{j=1}^{m} w_j + \sum_{j=1}^{k} t_j \geq \sum_{j=1}^{m} (w_j + t_j) > m$$

en dat is natuurlijk een tegenstrijdigheid.

Maar met deze twee opmerkingen volgt de stelling nu natuurlijk onmiddellijk: first fit dalend zou niet met een nieuwe vrachtwagen met index $j > m$ beginnen als op elk van de andere vrachtwagens met index $j > m$ niet al ten minste 3 gewichten zitten. Dus worden ten hoogste $\lceil \frac{m-1}{3} \rceil$ vrachtwagens met index $j > m$ gebruikt, dus samen met de eerste m zijn dat $m + \lceil \frac{m-1}{3} \rceil \leq \lceil \frac{4m}{3} \rceil$

■

Oefening 99 *Geef uitleg waarom de formule bij stap (*) ook voor best fit dalend klopt of hoe de argumenten gewijzigd moeten worden.*

Oefening 100 *Stel dat het grootste gewicht in de rij van gewichten die geplaatst moeten worden grootte $\frac{1}{x}$ heeft met $x > 3$. Bewijs een stelling die analoog is aan Stelling 26, maar een betere grens geeft dan $\lceil \frac{4}{3}m \rceil$. Bewijs de beste grens die je kan bewijzen.*

Oefening 101 *Geef voor elk van de volgende gevallen ofwel een voorbeeld of toon aan dat een dergelijk voorbeeld niet bestaat:*

- *first fit presteert beter dan best fit*
- *best fit presteert beter dan first fit*
- *first fit (online) presteert beter dan first fit dalend (offline)*
- *first fit dalend (offline) presteert beter dan first fit (online)*
- *best fit (online) presteert beter dan best fit dalend (offline)*
- *best fit dalend (offline) presteert beter dan best fit (online)*

Oefening 102 *Beschrijf expliciet de details van een goed algoritme dat best fit dalend implementeert en geef de tijdscomplexiteit.*
Met expliciet *wordt bedoeld dat je vooral expliciet zegt hoe je de juiste vrachtwagen bepaalt om het gewicht te plaatsen.*

9 Gerandomiseerde algoritmen

Misschien zou het voor dit gedeelte nuttig zijn jullie basiskennis over kansrekenen nog eens op te frissen. Maar meer kennis dan de basisbegrippen is werkelijk niet nodig...
Jullie hebben al gezien dat het soms een voordeel kan zijn gerandomiseerd te werken omdat sommige algoritmen goed werken voor *toevallige* inputs maar problemen hebben met inputs die een zekere structuur hebben. Een voorbeeld is quicksort waarbij je als pivot altijd het eerste element kiest. Dat presteert heel slecht als het op reeds gesorteerde rijen toegepast wordt maar het presteert heel goed op ordeningen die geen bijzondere structuur vertonen. Dus kan je in gevallen waar de mogelijkheid bestaat dat de input al gedeeltelijk gesorteerd is de performantie verbeteren door op de rij die gesorteerd moet worden eerst een toevallige permutatie toe te passen en pas daarna quicksort uit te voeren! (Of je besteedt iets meer aandacht aan de keuze van de pivot...)
Wij zullen nu twee soorten algoritmen zien die nog een stap verder gaan:

9.1 Las Vegas Algoritmen

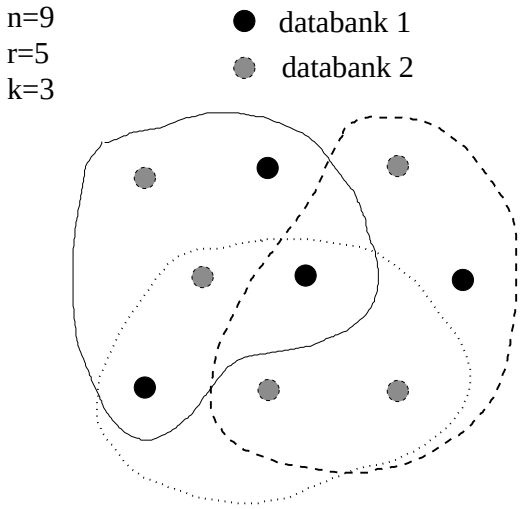
Las Vegas Algoritmen zijn algoritmen waar het toeval meespeelt maar waar het antwoord altijd juist is. De invloed van het toeval is op de uitvoeringstijd. Het principe van Las Vegas algoritmen is meestal hetzelfde:

- Probeer een oplossing te raden.
- Test of wat je geraden hebt een oplossing van jouw probleem is.

Je kan het vaak ook zo zien dat je een een recursief algoritme – bv. branch and bound – neemt, maar in plaats van exhaustief alle vertakkingen te doorlopen alleen één vertakking neemt die toevallig gekozen is. Belangrijk is dat je kan bewijzen dat op deze manier een goede kans bestaat een oplossing te vinden. De manier van werken versta je het best als je naar een voorbeeld kijkt.
In ons voorbeeld zullen wij zien dat hoewel er geen bovengrens is voor de uitvoeringstijd de *verwachte* uitvoeringstijd heel goed is.

Voorbeeld: Gegeven een verzameling van computers $V = \{C_1, \ldots, C_n\}$ en k groepen van gebruikers, waarbij sommige gebruikers in meerdere groepen kunnen zitten. Groep i kan computers in de deelverzameling $V_i \subseteq V$ gebruiken. Er is bovendien een grens $r \geq 2$ zodat elke verzameling V_i een grootte

van ten minste r heeft en wij weten dat er ten hoogste $k \leq 2^{r-3}$ groepen van gebruikers zijn.
Ons doel is op elke computer één van 2 databanken te installeren zodat in elke groep van gebruikers elke databank gebruikt kan worden – of met andere woorden: zodat in elke V_i ten minste één kopie van elke databank is.



Figuur 39: Een klein voorbeeld voor ons databankverdelingsprobleem.

Normaal zouden wij hiervoor een branch and bound algoritme toepassen. Het probleem hierbij is dat het zou kunnen gebeuren dat het algoritme heel lang moet zoeken omdat het in een tak van de recursieboom begint te zoeken waar er geen oplossing zit. Daarom gebruiken wij een Las Vegas algoritme:

- herhaal de volgende stappen totdat een oplossing is gevonden:
  - Kies onafhankelijk met kans $\frac{1}{2}$ voor elke computer in V één van de twee databanken.
  - Test of deze verdeling aan de vereisten voldoet.

Als er geen oplossing is, draait dit algoritme dus oneindig lang.
Een lus kan zeker in lineaire tijd uitgevoerd worden (lineair in $n + k$ als wij r als vast beschouwen).
Maar hoe groot is de kans een oplossing te vinden? Wij hebben een oplossing gevonden als er in geen van de verzamelingen alleen maar databank 1 zit of alleen maar databank 2.
De kans dat een zekere computer databank 1 krijgt, is $\frac{1}{2}$. De kans dat alle $r' \geq r$ computers in een vaste verzameling databank 1 krijgen, is dus $(\frac{1}{2})^{r'} \leq$

$(\frac{1}{2})^r$. Hetzelfde geldt natuurlijk voor databank 2. De kans dat een vaste verzameling alleen maar databank 1 of alleen maar databank 2 bevat, is dus ten hoogste $2*(\frac{1}{2})^r$. Als wij nu kijken hoe groot de kans is dat ten minste één van de $k \leq 2^{r-3}$ verzamelingen deze eigenschap heeft dan is dat ten hoogste $2^{r-3} * 2 * (\frac{1}{2})^r = \frac{1}{4}$. Dat is dus ten hoogste de kans dat wij na één iteratie nog **geen** oplossing hebben. Dus is de kans dat wij na één iteratie **wel** een oplossing gevonden hebben ten minste $1 - \frac{1}{4} = \frac{3}{4}$.
En inderdaad toont dat ook dat er een oplossing **is**. Als er geen oplossing was, zou het natuurlijk onmogelijk zijn er een te vinden – de kans zou dus 0 zijn.
De kans dat er na p pogingen nog geen oplossing is gevonden, is dan ten hoogste $(\frac{1}{4})^p$ en het verwachte aantal pogingen dat nodig is, is 4/3. **Maar:** Voor elke p is de kans om meer dan p pogingen nodig te hebben positief – hoewel de verwachte tijd dus heel klein is, is er geen bovengrens voor het slechtste geval.

Het feit dat zelfs een toevallige verdeling kans $\frac{3}{4}$ heeft om een oplossing te zijn, wijst erop dat het in zekere zin een gemakkelijk probleem is. Zou het dus ook voor dit geval kunnen gebeuren, dat een branch and bound algoritme veel tijd vraagt?
Stel bv. dat alle oplossingen de eigenschap hebben dat computer 1 en computer 2 niet allebei databank 1 hebben. Dan is het nog altijd mogelijk dat drie kwart van alle verdelingen een oplossing vormen. (Of dat voor dit voorbeeld **echt** kan gebeuren, is niet zo belangrijk – het gaat erom te verstaan waar Las vegas algoritmen voordelen in vergelijking met branch and bound kunnen hebben.)
Als je dan een branch and bound algoritme hebt dat met databank 1 voor deze computers begint, moet je de hele deelboom afwerken, voordat je in de deelboom terecht komt waar de goede oplossingen zitten – en dat kan heel lang duren. Ook voor een branch and bound algoritme is de kans groot dat die snel een oplossing vindt. Maar als het algoritme een slechte keuze maakt, kan het best heel lang duren voordat de deelstructuur die het belet een oplossing te vinden terug verwijderd wordt. Het Las Vegas algoritme kan ook een slechte keuze maken – maar doordat altijd helemaal opnieuw een oplossing wordt opgebouwd, zal een slechte keuze maar een heel korttijdig effect hebben!

Oefening 103 *Gegeven een constante r.*
In een computernetwerk met ten hoogste $|V| = \frac{1}{4} * (\frac{3}{2})^r$ computers moeten kopieën van drie verschillende databanken A, B, C zo op de computers geplaatst worden dat voor elke databank $d \in \{A, B, C\}$ elke knoop d bevat of

een directe buur heeft die d bevat. Elke computer mag maar één databank bevatten. Je weet dat elke knoop tenminste r directe buren heeft. Geef een Las Vegas Algoritme dat met kans ten minste $\frac{1}{2}$ na één poging een oplossing heeft gevonden. Eén poging moet in lineaire tijd uitgevoerd kunnen worden (lineair in het aantal computers plus het aantal verbindingen tussen de computers).

Oefening 104 *Aan de universiteit bestaan verschillende commissies waarbij sommige mensen natuurlijk tot meerdere commissies kunnen behoren. Nu worden 4 bijscholingscursussen aangeboden en het probleem is, veilig te stellen dat aan de ene kant elke mens precies één van de cursussen volgt maar aan de andere kant de expertise uit elke cursus ook in elke commissie beschikbaar zal zijn (dus moet voor elke cursus gelden dat in elke commissie iemand zit die de cursus volgt).*
Als bv. een commissie maar 3 mensen bevat is dat natuurlijk niet mogelijk, maar stel vanaf nu dat elke commissie ten minste $c \geq 5$ leden bevat en dat er minder dan $4^{c-1}/3^c$ commissies zijn.

a.) *Stel dat je alleen maar weet dat elke commissie tenminste 10 personen bevat en dat er ten hoogste 4 commissies zijn (dat voldoet aan de voorwaarde), maar je weet niet welke personen tot welke commissie behoren. Is dat voldoende om te beslissen of er een oplossing voor het probleem bestaat ? Toon aan dat jouw antwoord juist is.*

b.) *Geef een Las Vegas Algoritme dat met positieve kans na één poging in lineaire tijd een oplossing kan hebben gevonden.*

Oefening 105 *In DA III zullen jullie zien dat je niet alle bestanden kan comprimeren maar dat je inderdaad ten hoogste 1/255 van alle mogelijke bestanden met een gegeven aantal bytes kan comprimeren. Voor deze oefening mag je daarvan uitgaan.*
Als je het algoritme van gzip kent, kan je natuurlijk jouw kennis gebruiken om een bestand te vinden dat niet gecomprimeerd kan worden. Maar stel nu dat je een compressiealgoritme hebt waarvan je wel weet dat het heel snel is (bv. in lineaire tijd een bestand kan comprimeren) maar niet wat het precies doet.
De taak is een efficiënt algoritme te beschrijven dat voor dit compressiealgoritme een niet comprimeerbaar bestand met n bytes kan vinden. Inderdaad lijkt hier een gerandomiseerd algoritme geschikt te zijn.

- *Beschrijf een Las-Vegas algoritme voor dit probleem.*
- *Hoe groot is de kans dat dit algoritme meer dan 3 iteraties nodig heeft om het bestand te vinden?*

9.2 Monte Carlo Algoritmen

Voor Monte Carlo Algoritmen is de tijd (meestal) niet afhankelijk van het toeval – maar het resultaat kan er wel van afhankelijk zijn en soms kan het zelfs fout zijn. Op het eerste gezicht lijkt het gek algoritmen te beschouwen die een fout antwoord kunnen geven – maar als we eerlijk zijn dan is ook in het geval van gegarandeerd juiste algoritmen de kans op een implementatie met een fout – dus programma's die een fout antwoord geven – toch niet te verwaarlozen. Goede Monte Carlo algoritmen hebben een **heel** kleine kans op een foutief antwoord voor een gegeven invoer – zelfs zo klein dat misschien de kans van een foutieve implementatie van een algoritme dat *in principe* een juist antwoord garandeert maar veel ingewikkelder is om te implementeren groter is. In sommige gevallen kan dus de kans dat een programma een fout antwoord geeft kleiner zijn als je een algoritme implementeert dat soms fouten maakt...
Een kleine kans op een fout antwoord betekent dat de kans op een fout voor **elke** invoer **bewijsbaar** klein is – dus niet dat er invoeren zijn waarvoor het antwoord altijd fout is en dat de kans op zo'n invoer klein is.
Maar het belangrijkste voordeel is dat Monte Carlo algoritmen soms veel sneller zijn dan gegarandeerd juiste algoritmen en tegelijk een verwaarloosbaar kleine kans op een fout hebben.
Ook hier is de beste kennismaking gewoon naar voorbeelden te kijken. Eerst zullen wij een gemakkelijk voorbeeld zien dat vooral het principe aantoont en dan nog een belangrijk voorbeeld, nl. priemgetaltesten.
Voorbeeld: Wij hebben een random number generator waarvan wij weten dat hij getallen uit $0, \ldots, x$ gelijkverdeeld genereert – maar wij kennen x niet. Nu zullen wij een algoritme beschrijven dat een goede bovengrens voor x bepaalt – of precies: een x' waarvoor geldt dat $x \leq x' \leq 1.05 * x$. Dit algoritme zal een Monte Carlo algoritme zijn waarvan het antwoord niet altijd juist is. Voor het antwoord x' zal **altijd** gelden dat $x' \leq 1.05 * x$ maar met een heel kleine kans zou het kunnen gebeuren dat wij een x' kiezen dat kleiner dan x is. Het algoritme werkt als volgt (waarbij m een vast getal is dat op voorhand vastgelegd is):

- genereer m toevallige getallen met deze generator.
- kies het maximum x_0.
- voer $x' = 1.05 * x_0$ uit.

Het is duidelijk dat x' nooit groter kan zijn dan $1.05 * x$, maar hoe groot is de kans dat $x' < x$? Dat gebeurt als $x_0 < \frac{20}{21} * x$. De kans dat één toevallige

uitvoer y van de generator voldoet aan $y < \frac{20}{21} * x$ is ten hoogste $\frac{20}{21}$. De kans dat het maximum (dus alle uitvoeren) kleiner zijn dan $\frac{20}{21}$ is dus ten hoogste $\frac{20}{21}^m$. Als wij m zo groot kiezen dat $\frac{20}{21}^m < \frac{1}{10^{20}}$ (daarvoor zou $m \geq 944$ moeten zijn) is dat zeker een aanvaardbare kans op een fout en het algoritme zou inderdaad heel snel zijn en zelfs onafhankelijk van x! De uitvoeringstijd is dus in feite constant!

9.2.1 De Miller Rabin priemgetallentest

De Miller Rabin priemgetallentest is een voorbeeld van een Monte Carlo algoritme dat in de praktijk bijzonder belangrijk is. Het zoeken en testen van priemgetallen is bv. een deel van het RSA (Rivest Shamir Adleman) cryptosysteem. In principe lijkt het algoritme heel sterk op het gemakkelijke voorbeeld dat wij net hebben gezien – maar de redenering die nodig is om de kans van een fout antwoord in te schatten, is duidelijk moeilijker.
Voor de Miller Rabin priemgetallentest worden twee stellingen uit de getallentheorie gebruikt

Stelling 27 *(De kleine stelling van Fermat)*
Als p een priemgetal is en $0 < a < p$ dan is $a^{p-1} \equiv 1(mod\ p)$.

Voorbeeld:
$p = 7$, $a = 4$, $4^{7-1} = 4096$ en $4096 = 585 * 7 + 1$ dus $4096 \equiv 1(\text{mod } 7)$.

Als wij dus een getal x willen testen om te weten of het een priemgetal is en voor een toevallig gekozen getal a met $0 < a < x$ geldt $a^{x-1} \not\equiv 1(\text{mod } x)$ dan is x zeker geen priemgetal. Als het resultaat zou zijn $a^{x-1} \equiv 1(\text{mod } x)$ helpt dat niet veel: misschien is x een priemgetal en misschien niet...
Er is nog een tweede stelling die kan helpen:

Stelling 28 *Als p een priemgetal is dan zijn de enige oplossingen voor $x^2 \equiv 1(mod\ p)$ met $x \in \{0, \ldots, p\}$ de getallen 1 en $p - 1$.*

Bewijs: $x^2 \equiv 1(\text{mod } p) \Rightarrow x^2 - 1 \equiv 0(\text{mod } p) \Rightarrow (x+1)(x-1) \equiv 0(\text{mod } p)$
$\Rightarrow x \in \{1, p-1\}$ waarbij alleen voor de laatste stap gebruikt werd dat p een priemgetal is.

■

Als wij dus een getal x hebben met $1 < x < p - 1$ en wij stellen vast dat $x^2 \equiv 1(\text{mod } p)$ dan kunnen wij zeker zijn dat p geen priemgetal is.
Ook hier hebben wij dus de mogelijkheid soms met zekerheid te zeggen dat een getal geen priemgetal is – maar ook deze stelling kan niet gebruikt worden om aan te tonen dat een getal wel een priemgetal is.

Om de kleine stelling van Fermat te kunnen toepassen om een getal x te testen, hebben wij a^{x-1} nodig. Dat zullen wij op een manier berekenen waar wij veel kwadraten (waarop wij Stelling 28 kunnen toepassen) *gratis* erbij krijgen. Deze manier om een macht te berekenen, hebben jullie misschien al in een vroegere les gezien:

Stel dat wij a^k voor positieve gehele getallen a, k willen berekenen en dat $b_n, \ldots, b_0$ de binaire voorstelling van k is. Dan kan je dat op de volgende square and multiply manier doen:

```
pot=1;
for (i=n; i>=0; i--)
   { pot=pot*pot; // hier krijg je kwadraten gratis
     if (b_i=1) pot=pot*a;
   }
// a^k staat nu in de variabele pot
```

Oefening 106 *Stel dat je a^k voor positieve gehele getallen a, k met de square and multiply methode berekent. Hoeveel kwadraten worden tijdens deze methode berekend? Het antwoord moet een functie van k zijn.*

Dat gebruiken wij nu om te testen of een getal een priemgetal is. Stel dat wij een oneven getal $x > 3$ hebben en willen testen of het een priemgetal is.

Algoritme 4 *Miller Rabin priemgetallentest*

- *Herhaal de volgende stappen m keer of tot een waarde terug wordt gegeven:*
  - *kies een toevallig getal $0 < a < x$*
  - *bereken $a^{x-1}(mod\ x)$ met de square and multiply methode.*
    - *als voor één van de kwadraten die je tijdens de berekening tegenkomt, geldt dat $pot \notin \{1, x-1\}$ maar $pot^2 \equiv 1(mod\ x)$ geef* **niet priem** *terug.*
    - *als $a^{x-1} \not\equiv 1(mod\ x)$ geef* **niet priem** *terug.*
- *Geef* **priem** *terug.*

In dit algoritme wordt **altijd** modulo x gerekend – ook in de tussenstappen. Anders zou je met extreem grote getallen moeten werken en dat zou natuurlijk inefficiënt zijn!
Volgens de stellingen is het antwoord *niet priem* van dit algoritme altijd juist – maar het antwoord *priem* niet altijd. Wat is de kans op een fout als wij de binnenste lus m keer herhalen? Daarvoor eerst een definitie.

Definitie 19 *Gegeven een oneven getal x. Dan kunnen wij $x-1$ schrijven als $x-1=2^t u$ met een oneven getal u en $t \geq 1$.*
Als nu $b \in \{1,\ldots,x-1\}$ en
$b^u (mod\ x) \notin \{1,-1\}$ **en** $\forall 0 < s < t : b^{2^s u} \not\equiv -1 (mod\ x)$
dan noemen wij b een speciale getuige voor x.

De volgende oefening toont dat een speciale getuige in feite een getuige is van het feit dat x niet priem is.

Oefening 107

- *Wat is de samenhang tussen de getallen t en u in de definitie en de binaire voorstelling $b_n,\ldots,b_0$ van een binair getal $x-1$?*

  Bereken eerst de binaire voorstellingen, t en u van $x-1$ voor $x=9$ en $x=15$.

- *Gebruik de stellingen 27 en 28 om aan te tonen dat het bestaan van een getuige inderdaad impliceert dat x niet priem is.*

Voor deze *getuigen* is bewezen dat voor een oneven getal $x > 3$ dat geen priemgetal is ten minste $\frac{3}{4}$ van de getallen $1,\ldots,x-1$ getuigen zijn. Als ons algoritme dus voor dergelijke getuigen **niet priem** zou teruggeven (en wij weten al dat het juist is als het algoritme `niet priem` teruggeeft), dan zou de kans op een antwoord **priem** in een geval waar het geen priemgetal en wij de binnenste lus maar 1 keer doorlopen dus ten hoogste $\frac{1}{4}$ zijn. Als wij de test m keer onafhankelijk herhalen, zou de kans op een fout antwoord **priem** dus ten hoogste $\frac{1}{4^m}$ zijn – dus al voor $m=50$ verwaarloosbaar klein.
Nu moeten wij ons nog overtuigen dat ons algoritme opgestart met een getuige `a` inderdaad **niet priem** teruggeeft:
Stel dat a een getuige is. Als $a^{x-1} (\text{mod } x) \not\equiv 1 (\text{mod } x)$ wordt **niet priem** teruggegeven – dat is dus OK.
Als $a^{x-1} (\text{mod } x) \equiv 1 (\text{mod } x)$ is er een minimale $t_0 \leq t$ zodat $a^{2^{t_0} u} \equiv 1 (\text{mod } x)$. Omdat $a^{2^0 u} (\text{mod } x) \notin \{1, x-1\}$ (eigenschap van speciale getuigen) is deze $t_0 > 0$ en dan is $a^{2^{t_0-1} u} \not\equiv x-1 (\text{mod } x)$ (dat is ook een eigenschap van speciale getuigen). Dat $a^{2^{t_0-1} u} \not\equiv 1 (\text{mod } x)$ is gewoon de minimale keuze van t_0.

Dus bestaat er een t_0 met $a^{2^{t_0}u} \equiv 1 (\text{mod } x)$ en $a^{2^{t_0-1}u} (\text{mod } x) \notin \{1, x-1\}$ – maar $a^{2^{t_0}u}$ is één van de kwadraten die door de test worden getest – dus wordt voor de getuige **niet priem** teruggegeven!

Oefening 108 *Zijn er gevallen waar het antwoord van de Miller-Rabin-test "niet priem" is en waar dit antwoord niet gebaseerd is op een eigenschap die ook speciale getuigen moeten hebben?*

Deze test kan niet alleen gebruikt worden om getallen te testen maar ook om priemgetallen te vinden. Daarvoor helpt de volgende stelling:

Stelling 29 *Voor (niet te kleine) $n \in \mathbb{N}$ voldoet het aantal $\Pi(n)$ van priemgetallen $p \leq n$ aan $\Pi(n) \approx \frac{n}{\ln n}$*

En dit geldt al voor relatief kleine n. Als je dus gewoon toevallige getallen kiest en die dan op de eigenschap test of het priemgetallen zijn dan is de kans groot dat je niet te lang moet zoeken voordat je er één hebt gevonden.
Intussen is een algoritme gekend om in polynomiale tijd te testen of een getal een priemgetal is. Maar dit algoritme is veel ingewikkelder en hoewel het polynomiaal is ook duidelijk trager. Omdat de kans op een fout van de Miller Rabin test zo klein is is het dus nog altijd de eerste keuze om te testen of een getal een priemgetal is.
Eén van de belangrijkste toepassingen van priemgetallen is RSA encryptie (Rivest, Shamir, Edelman). Het is alleen maar een interessante toepassing (om te zien hoe belangrijk priemgetallen zijn) maar niet echt het onderwerp van deze les (maar wel van andere lessen die jullie volgen).

Oefening 109 *Een random number generator genereert alle gehele getallen $0, \ldots, x$ met dezelfde kans. De uitvoer van deze generators zijn bitstrings die de binaire voorstelling van het getal zijn. Wij weten dat $x = 2^n - 1$ voor een zekere n, maar wij kennen noch x noch n. Deze keer willen wij niet weten wat een goede bovengrens voor x is, maar wij willen precies weten wat x is. Geef een Monte Carlo algoritme dat met kans ten hoogste $1/(2^{50})$ een fout resultaat levert.*

Oefening 110 *Een computerprogramma voor een beslissingsprobleem dat bepaalt of een geheel getal een zekere eigenschap heeft of niet (de mogelijke antwoorden zijn dus* ja *of* neen*) wordt door een foute compiler gecompileerd en de uitvoerbare bestanden met deze foute programma's worden ook via het internet verdeeld. Maar er zijn ook kopieën van dit programma die juist zijn. Je zit nu aan een computer en moet beslissen of het programma betrouwbaar is. Je weet dat de foute versies van het programma voor een kwart van de*

getallen **soms** *een fout antwoord geven. Ook voor de getallen waar iets misloopt is de kans een juist antwoord te krijgen nog altijd* $2/3$*. Beschrijf een algoritme om met kans ten minste 99% te weten of het programma waarmee je moet werken juist is.*
Let op: Je hebt geen lijst van juiste antwoorden ter beschikking!

Oefening 111 *Gegeven een getal* n *waarvoor wij willen testen of het priem is en een basis* b*.*

a.) *Beschrijf de twee gevallen wanneer de Miller-Rabin test zegt dat een getal geen priemgetal is. Waarop zijn deze beslissingen gebaseerd?*

b.) *Stel dat het resultaat van de Miller-Rabin test is "***geen** *priemgetal". Hoe groot is de kans dat dit antwoord juist is?*

c.) *Pas de Miller-Rabin test toe op* $n = 25$ *en* $b = 18$*.*

d.) *Pas de Miller-Rabin test toe op* $n = 21$ *en* $b = 8$*.*

Index