Project Zeepbelbomen

Rien Maertens
2^{de} Bachelor Informatica

24 november 2015

Theoretische Vragen

Opgave 1. *Geen top in een bladzeepbel heeft een kind in een andere zeepbel.*

Bewijs. Stel we hebben een bladzeepbel α met een top T die een kind K heeft in zeepbel β. Neem n het aantal zeepbellen dat het pad van zeepbel α naar de wortel bevat. We kunnen de volgende twee gevallen onderscheiden:

Geval 1 Zeepbel β bevat een top zonder kinderen en is dus een bladzeepbel. Het pad van de wortel van de boom naar β gaat door α. Dit pad bevat dus $n + 1$ zeepbellen. Dit is tegenstrijdig met het gegeven dat het pad van de wortel naar alle bladzeepbellen evenveel zeepbellen bevat. Het gestelde is dus vals. Het is onmogelijk voor een bladzeepbel om een kind te hebben in een andere zeepbel.

Geval 2 Zeepbel β is geen bladzeepbel, maar bevat wel een pad naar een top zonder kinderen. Deze top is onderdeel van bladzeepbel γ. Het pad van γ naar de wortel van de zeepbelboom bevat $n + k$ zeepbellen, met k het aantal zeepbellen in het pad tussen β en γ. Dit is opnieuw tegenstrijdigi met het gegeven dat het pad van de wortel naar alle bladzeepbellen hetzelfde aantal zeepbellen bevat. Ook in dit geval kan een bladzeepbel geen kinderen in andere zeepbellen hebben.

□

Opgave 2. *Wat is de maximale diepte van een k-zeepbelboom met n sleutels?*

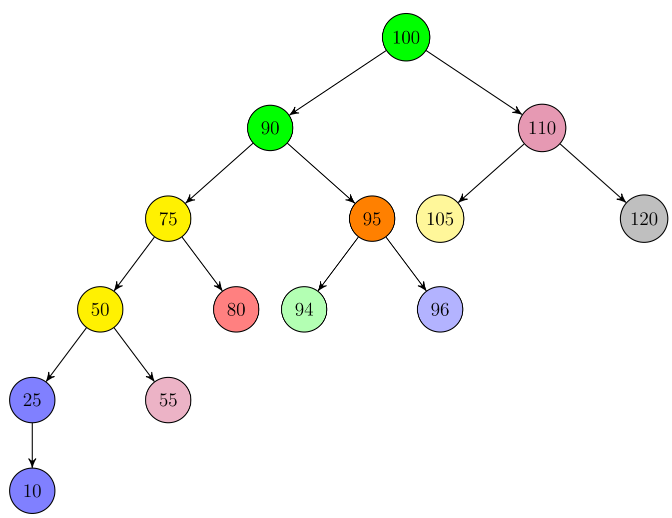
a. Gemeten in zeepbellen.

Stel d_z de diepte in zeepbellen van een zeepbelboom. d_z bereikt een maximum wanneer iedere zeepbel maar één top omvat, m.a.w. wanneer het pad van de wortel naar een blad evenveel zeepbellen als toppen bevat. De maximale diepte d_z in zeepbellen van een k-zeepbelboom met n sleutels is dus gelijk aan de maximale diepte in toppen van een complete binaire boom met n toppen. Dit is dus:

$$d_z \leq \lfloor \log_2 n \rfloor \tag{1}$$

b. Gemeten in toppen van de ten gronde liggende binaire boom.

Stel d_t de diepte in toppen van een zeepbelboom. d_t bereikt een maximum wanneer we één lang pad van zeepbellen maken waar elke top van een zeepbel maximum één kind in dezelfde zeepbel bevat. De andere zeepbellen die kinderen zijn van dit lange pad zijn terug zeepbellen met grootte één. Dit ziet er dan als volgt uit:



Het valt op dat wanneer we een niveau zeepbellen toevoegen, we het dubbele aantal toppen moeten toevoegen van de vorige zeepbelboom met een niveau lager. Voor n het aantal toppen en z het aantal niveaus in zeepbellen geldt:

$$n \geq \sum_{i=1}^{z} k2^{z-1} = k(2^z - 1)$$

We lossen dit op naar z:

$$z \leq \log_2 \left(\frac{n+k}{k} \right)$$

De maximale diepte d_t van deze zeepbelboom in toppen is gelijk aan het aantal niveaus zeepbellen vermenigvuldigd met het maximale aantal toppen per zeepbel. Dus krijgen we dat $d_t = k \cdot z$. Wanneer voegen dit alles samen tot één formule:

$$d_t \leq k \cdot \log_2 \left(\frac{n+k}{k} \right) \tag{2}$$

Waar d_t gelijk is aan de maximale diepte in toppen.

Opgave 3. *Analyse van de verschillende gebalanceerde zeepbelbomen en hun complexiteit.*

a. Voor toevoegen.

Het toevoegen van een item aan een gebalanceerde zeepbelboom gebeurt als volgt:

Eerst wordt het item opgezocht in de zeepbelboom. Wanneer het item gevonden werd dan zit het item al in de zeepbelboom en moet er niets gebeuren. In dit geval heeft deze bewerking dezelfde tijds- en geheugencomplexiteit als het opzoeken, respectievelijk $\mathcal{O}(\log n)$ en $\Theta(1)$ (dit bespreek ik in punt b).

Als de opzoeking uitkomt op `null` dan zit het toe te voegen item niet in de boom. In dit geval wordt er een nieuwe top aangemaakt en bevestigd aan de top die het laatst werd bekeken bij de zoekopdracht. De nieuwe top wordt ook toegevoegd aan de zeepbel waarin de ouder zit. Indien de zeepbel door deze toevoeging teveel toppen bevat wordt deze gesplitst. De manier waarop de zeepbel gesplitst wordt verschilt per implementatie van zeepbelboom:

- **Zeepbelboom1** zal zijn zeepbellen splitsen door eerst te kijken of beide kinderen van de wortel in dezelfde zeepbel zitten. Is dit niet het geval dan wordt een rotatie met de bovenste drie zeepbellen uitgevoerd zodat dit wel zo is. Vervolgens wordt de zeepbel gesplitst zodat elk kind van de oude wortel de wortel wordt van een nieuwe zeepbel. De oude wortel wordt daarna toegevoegd aan de zeepbel van zijn ouder, die eventueel opnieuw wordt gesplitst indien deze nu ook overvol zit. Als de oude wortel ook de wortel van de zeebelboom is dan wordt een nieuwe zeepbel aangemaakt.

Er wordt enkel binnen dezelfde zeepbel gekeken en omdat de groote van een zeepbel constant is, gebeurt het splitsen van een zeepbel in constante tijd. In het slechtste geval wordt iedere zeepbel vanaf de toegevoegde top tot aan de wortel gesplitst In opgave 2a hebben we bewezen (2) dat het aantal zeepbellen van de wortel naar een blad maximaal $\lfloor \log_2 n \rfloor$ is.

We kunnen dus concluderen dat voor Zeepbelboom1 met n toppen de toevoegbewerking $\mathcal{O}(n)$ tijd in beslag neemt en $\Theta(1)$ geheugen nodig heeft.

- **Zeepbelboom2** splitst door eerst de zeepbelboom zo goed mogelijk te balanceren. Hiervoor worden eerst alle toppen in gesorteerde volgorde in een lijst gestoken en vervolgens opgebouwd tot een zo compleet mogelijke binaire boom. Daarna wordt net zoals Zeepbelboom1 de wortel toegevoegd aan de zeepbel van zijn ouder en vormen de kinderen van deze wortel de wortels van de nieuwe zeepbellen.

  Net zoals Zeepbelboom1 wordt er enkel binnen dezelfde zeepbel gekeken, echter zal deze zeepbelboom er net iets langer over doen aangezien altijd alle toppen binnen de zeepbel wordt bekeken in plaats van enkel de bovenste drie. We besluiten dus dat een top toevoegen aan Zeepbelboom2 $\mathcal{O}(n)$ tijd in beslag neemt en $\Theta(1)$ geheugen nodig heeft voor een boom van n toppen.

- **Zeepbelboom3** balanceerd net zoals Zeepbelboom2 eerst de toppen in de te grote zeepbel, maar in plaats van enkel de wortel toe te voegen aan de bovenliggende zeepbel worden er meerdere toppen toegevoegd. Hoeveel toppen er toegevoegd worden kan worden aangepast. Als deze zeepbelboom werd gemaakt met de normale constructor da zullen er zoveel mogelijk toppen worden toegevoegd aan de bovenliggende zeepbel tot deze zijn maximum capactiteit heeft bereikt (maar niet te vol zit). Met een tweede constructor kan er ingesteld worden dat er een maximum staat op het aantal omhoog te duwen toppen en kan dit maximum ook meegegeven worden. Er kan ook ingesteld worden dat er, indien mogelijk, net zoveel toppen aan de bovenliggende zeepbel worden toegevoegd tot deze moet splitsen.

Hier geldt hetzelfde als de vorige twee zeepbelbomen: er wordt enkel in dezelfde zeepbel gekeken, wat in constante tijd gebeurt omdat de grootte van de zeepbel constant is. Een splitsing kan een kettingreactie veroorzaken die bovenliggende zeepbellen ook doet splitsen. Maar er zullen maximum $\mathcal{O}(\log n)$ van deze splitsingen geberuren. De toevoegbewerking van Zeepbelboom3 zal dus ook $\mathcal{O}(\log n)$ tijd en $\Theta(1)$ geheugen nodig hebben voor een boom waar reeds n elementen in zaten opgeslagen.

b. Voor opzoeken.

De drie gebalanceerde zeepbelbomen gebruiken alledrie dezelfde opzoekmethode, dus de complexiteit van deze bewerking is dezelfde. Deze methode werkt zoals een normale binaire zoekboom:

De wortel wordt vergeleken met het gezochte item. Is het item in de wortel kleiner dan dalen we af naar het rechterkind van de wortel. Is het item in de wortel groter dan dalen we af naar het linkerkind van de wortel. Komt de waarde van de wortel overeen met dat van het gezochte item dan hebben we het item gevonden en kunnen we terugkeren uit de functie. We blijven op deze manier zoeken tot we het gezochte item gevonden hebben of tot we niet meer verder kunnnen afdalen in de zoekboom (als we null tegenkomen). In dit geval zit het gezochte item niet in de zeepbelboom.

(a) Tijdscomplexiteit

In het slechtste geval zit het item dat wordt gezocht niet in de zeepbelboom en moeten we helemaal tot een blad van de boom afdalen. In (2) heb ik de maximale diepte in toppen berekend. In die formule is k is een constante dus valt die in asymptotische notatie weg. We kunnen dus besluiten dat het opzoeken van een element in een gebalanceerde zeepbelboom die n items bevat een tijdscomplexiteit van $\mathcal{O}(\log n)$ heeft.

(b) Geheugencomplexiteit

De implementatie van de zoekbewerking maakt gebruik van de `find()` methode. Deze maakt gebruik van een `while`-loop om in de boom af te dalen en maakt geen gebruik van andere datastructuren. De geheugencomplexiteit om iets op te zoeken in een gebalanceerde binaire boom is dus $\Theta(1)$.

Opgave 4. *Bepaal voor semi-splay zeepbelbomen de slechtste-geval complexiteit en de geamortiseerde complexiteit.*

Stelling 1. *Het langste pad van de wortel naar een blad in een semi-splay zeepbelboom is naar boven begrens door* $\mathcal{O}(\log(n)$

Bewijs. Het slechtste geval doet zich voor wanneer alle zeepbellen op één lijn staan. De lengte van het pad van de wortel naar de top die zich het verst van de wortel bevind is dan gelijk aan het aantal zeepbellen vermenigvuldigd met de maximale diepte van een zeepbel.

Het aantal zeepbellen z in een zeepbelboom met n toppen en maximaal k toppen per zeepbel is gelijk aan $\lfloor n/k \rfloor$. Aangezien een complete zeepbel altijd gebalanceerd is heeft deze diepte $\log_2(k)$. Het langst mogelijke pad van de wortel naar een blad is $\lfloor n/k \rfloor \cdot \log_2(k) = \mathcal{O}(n)$ toppen lang.

In asymptotische notatie wordt dit $\mathcal{O}(\log(n)$. Dit is precies wat we zochten. □

Stelling 2. *Een semi-splay bewerking gebeurt in constante tijd.*

Bewijs. Wanneer een splaybewerking wordt uitgevoerd worden alle toppen uit de drie geselecteerde zeepbellen in inorde overlopen en in een lijst gestopt. Bij het inorde itereren worden er $3 \cdot k$ toppen overlopen, me k de maximale grootte van een zeepbel. Vervolgens wordt deze lijst opgedeeld in drie lijsten van lengte k en worden van deze lijsten drie complete gebalanceerde zeepbellen gemaakt. Deze drie zeepbellen worden daarna aan elkaar bevestigd en terug in de zeepbelboom geplaatst.

Omdat er altijd binnen de gegeven drie zeepbelle wordt gekeken zal een semisplay bewering altijd $\Theta(k)$ operaties op toppen nodig hebben. Maar omdat de k in een zeepbelboom constant is kunnen we zeggen dat een splaybewerking in constante tijd of $\Theta(1)$ tijd verloopt. □

a. Voor toevoegen.

b. Voor opzoeken.

- **Slechtste-geval** Het slechtste geval doet zich voor wanneer het gezochte item zich het verst van de wortel bevind. In stelling 1 hebben we bewezen dat dit langste pad naar boven begrensd wordt door $\mathcal{O}(n)$. Wanneer dit geval is moeten alle toppen over dit pad overlopen worden om het gezochte item te vinden. Nadat het item werd gevonden moet er nog gesplayd worden tot aan de wortel.

Implementatie

Hieronder volgt er een korte uitleg over de interne structuur en algoritmes die ik heb gebruikt voor de implementaties van de zeepbelbomen.

1 Binaire boom

1.1 Top

Mijn binaire boom bestaat intern uit toppen die hun ouder, linker en rechterkind bijhouden. Om problemen met deze links te vermijden kan de verwijzing naar de ouder van een top enkel ingesteld worden wanneer deze top als kind wordt aangeduid van een andere top met de methodes `setRightChild()` en `setLeftChild()`. Omdat de wortel van de binaire boom natuurlijk geen ouder heeft is er ook een methode die deze verwijzing op `null` zet. Een top bevat ook een booleaanse waarde `removed` die functioneert als grafsteen. Daarnaast bevat iedere top ook een verwijzing naar de zeepbel waartoe die top behoort.

1.2 Zeepbel

Om de toppen in groepjes in te delen bestaat de klasse `Zeepbel`. Deze houdt enkel zijn wortel bij en het aantal toppen dat deze bezit. Een zeepbel heeft ook de methodes `topAdded()` en `topsRemoved()` die de zeepbel waarschuwen wanneer er toppen bijgekomen of weggevallen zijn. De methode `topAdded()` heeft ook een boolean terug die zegt of de zeepbel te groot is en verkleind moet worden.

1.3 Zeepbelboom

In de abstracte klasse Zeepbelboom zit de gemeenschappelijke functionaliteit die alle zeepbelboom-implementaties delen. Het is enkel de methode `shrinkBubble()` die verschilt tussen de verschillende zeepbelbomen. Deze methode zorgt ervoor dat een overvolle zeepbel gesplitst wordt.

2 Implementaties zeepbelboom

2.1 Zeepbelboom1

De eerste zeepbelboom lost een overvolle zeepbel op door zo weinig mogelijk veranderingen te brengen aan de interne structuur van de boom. Wanneer de wortel van de overvolle zeepbel beide kinderen in diezelfde zeepbel heeft dan worden er geen wijzigingen aangepast aan de interne structuur: deze twee kinderen vormen de wortels voor twee nieuwe zeepbellen en de top wordt toegevoegd aan de bovenliggende zeepbel. Als de wortel van de zeepbel maar één kind in dezelfde zeepbel bevat worden de eerste drie toppen geroteerd zodat de wortel wel twee kinderen in dezelfde zeepbel heeft en er kan gesplitst worden zoals normaal.

2.2 BalancingBubbleTree

Dit is een superklasse voor alle zeepbelbomen die een zeepbel moeten kunnen balanceren. Deze balancering gebeurt door de methode `balanceBubble()`. Dit gebeurt door eerst alle toppen in inorde te overlopen en in een lijst te plaatsen. Vervolgens wordt van deze lijst een nieuwe binaire boom opgebouwd door de recursieve methode `listToTree()`. Dit balanceren gebeurt in $\Theta(n)$ tijd als we het aantal te balanceren toppen als n kiezen. Maar er wordt enkel in één zeepbel gebalanceerd en een zeepbel heeft een maximaal aantal toppen, dus werkt deze methode in constante tijd.

2.3 Zeepbelboom2

Voor de zeepbel gesplitst wordt wordt deze eerst gebalanceerd met de methode uit `BalancingBubbleTree`. Daarna wordt op dezelfde manier gesplitst als `Zeepbelboom1`.

2.4 Zeepbelboom3

Deze zeepbel werkt op dezelfde manier als `Zeepbelboom2` echter worden er hier zoveel mogelijk toppen aan de bovenliggende zeepbel toegevoegd. Deze zeepbelboom kan geconfigureerd worden: er kan gekozen worden om een maximum te zetten op het aantal omhoog te duwen toppen en er kan ingesteld worden dat er geprobeerd wordt om net genoeg toppen toe te voegen aan de bovenliggende zeepbel dat die verplicht wordt om te splitsen.

3 Functies

3.1 Iterator

Een top heeft een methode `traverseInorder()` waarmee er in inorde kan geïtereerd worden over zijn kinderen. Aan deze methode worden twee functionele interfaces meegegeven: een `consumer` waaraan het volgende element in de iteratie wordt aan doorgegeven en een `predicate` die extra restricties kan opleggen over welke toppen er geïtereerd mag worden (bijvoorbeeld enkel over de toppen van een bepaalde zeepbel. Alle iterators maken van deze methode gebruik.

3.2 `Zeepbelboom.find()`

De drie basisoperaties: `add()`, `contains()` en `remove()` maken gebruik van de methode `find()` om een top op te zoeken. Deze methode werkt door binair te zoeken naar een `Top` tot de juiste top gevonden is of tot er `null` werd gevonden, waarna de top aan de juiste consumer wordt doorgegeven.

3.3 `Zeepbelboom.add()`

Er wordt eerst met `find()` gezocht naar de juiste top, als de top werd gevonden dan gebeurt er niets en wordt `false` teruggegeven. Als de top niet werd gevonden wordt top toegevoegd op de plaats waar `null` werd tegengekomen en wordt de betreffende zeepbel verwittigd van deze toevoeging, waarna er eventueel kan gesplitst worden als deze zeepbel overvol is.

3.4 `Zeepbelboom.contains()`

Hier wordt enkel met `find()` gezocht en teruggegeven of de top gevonden werd of niet.

3.5 `Zeepbelboom.remove()`

De te verwijderen top wordt opnieuw gezocht met `find()`. Wanneer de top gevonden werd wordt er een grafsteen geplaatst door de `removed` waarde op `true` te zetten. Als er een kritiek aantal grafstenen wordt bereikt (standaard 50%) wordt de boom opnieuw opgebouwd zonder de grafstenen.

Ik was begonnen met een implentatie die de toppen echt uit de zeepbelboom verwijdert, maar helaas zitten er nog wat bugs in de code en had ik geen tijd meer om deze te debuggen. Dus heb ik een werkende implementatie met grafstenen gemaakt.

Experimenten

Nu volgen de resultaten van enkele testen die ik heb uitgevoerd op de zeepbelbomen.