Project Treaps

Maksym Ziborov

01-12-2025

Inhoudsopgave

1 Theoretische vragen

Voor bijna alle vragen over de complexiteit van Treaps baseer ik mij op Aragon & Seidel (1989), Randomized Search Trees.

Vraag 1. *Toon aan dat de prioriteiten van een treap zo gekozen kunnen worden dat de diepte van de treap lineair is in het aantal toppen.*

Kies prioriteit op de volgende manier:

1) Kies de prioriteit van de eerste toegevoegde top willekeurig.
2) Voor elke andere top, kies de prioriteit als de prioriteit van de ouder na binaire toevoeging, verhoogd met één.

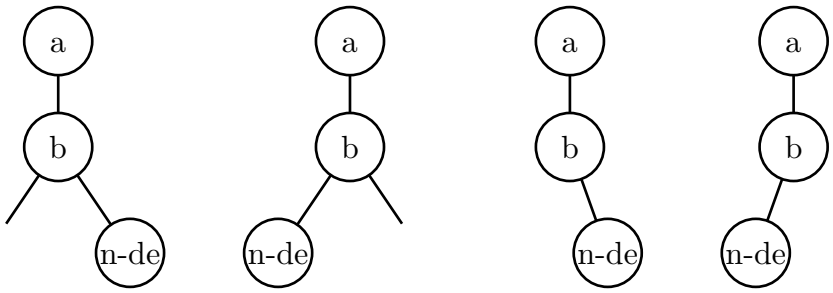
Bewijs door inductie op het aantal toppen n.

Basisgeval: Als $n = 1$, dan is de diepte ook 0, en dus lineair ten opzichte van het aantal toppen.

Inductiehypothese: Stel dat voor een boom met $n-1$ toppen, waarbij de prioriteit van elke top op bovenstaande manier is toegekend, geldt dat de diepte $n-2$ is.

Te bewijzen: Een boom met n toppen, waarbij de prioriteit van elke top op bovenstaande manier is toegekend, heeft diepte $n-1$.

Aangezien de boom voor de toevoeging van de n-de top volgens de inductiehypothese een diepte van $n-2$ heeft, belandt de toevoeging van de n-de top ons in een van de volgende gevallen:



Figuur 1: Mogelijke gevallen bij het toevoegen van de n-de top.

In alle gevallen is één rotatie van **Figuur 6** genoeg om de heap-eigenschap te herstellen. Na de rotatie is de resulterende boom de oorspronkelijke boom met diepte $n-2$, met één top toegevoegd op het pad van de wortel naar het blad, wat een diepte van $n-2+1 = n-1$ oplevert, waarmee het gestelde bewezen is.

Vraag 2. *Beschrijf de toevoegingsoperatie van een treap en bereken de tijdscomplexiteit.*

Toevoegoperatie gebeurt volgens volgend algoritme:

1. Voeg sleutel binair toe. Als sleutel al in boom ziet, stop met procedure, anders kies willekeurige prioriteit p voor de top.
2. Terwijl p groter is dan prioriteit van ouder, voer rotatie uit zoals beschreven in **Figuur 6**. Rechtse rotatie als toegevoegde top linkse kind is en anders een linkse rotatie.

Als d de diepte van de boom is, dan is de complexiteit in het slechtste geval $O(2d)$ aangezien wij in het slechtste geval tot aan een blad moeten doorlopen om binair toe te voegen en daarna de top terug naar de wortel moeten roteren om de heap-eigenschap te herstellen. Aangezien de diepte van een treap bij willekeurige keuze van prioriteit voor elke top $\log n$ blijft, krijgen wij een complexiteit van $O(2 \log n) = O(\log n)$ voor de toevoegoperatie.

Vraag 3. *Gegeven twee treaps T_1 en T_2 waarbij de grootste sleutel van T_1 kleiner is dan de kleinste sleutel van T_2. Geef een algoritme dat als invoer T_1 en T_2 neemt en een nieuwe treap T teruggeeft die de sleutels van beide treaps bevat en de prioriteiten bewaart.*

```
larger(n1, n2):
  return n2 == null || n1.getPriority() > n2.getPriority()

merge(root, child):
  if root == null || child == null: return
  if root.getValue() > child.getValue():
    left = true
    next = root.getLeft()
  else:
    left = false
    next = root.getRight()
  if larger(child, next):
    if left:
      root.setLeft(child)
    else:
      root.setRight(child)
    merge(child, next)
  else:
    merge(next, child)

merge(T1, T2):
  if larger(T1.root(), T2.root()):
    merge(T1.root(), T2.root())
    return T1
  else:
    merge(T2.root(), T1.root())
    return T2
```

Vraag 4. *Toon aan dat twee treaps steeds isomorf zijn als ze dezelfde sleutels met dezelfde prioriteiten bevatten en zowel de sleutels als de prioriteiten onderling verschillend zijn.*

Om in het bewijs niet elke keer de twee eigenschappen te moeten uitschrijven, geef ik hen een nummer:

1. Twee treaps bevatten dezelfde sleutels met dezelfde prioriteiten.
2. Zowel de sleutels als de prioriteiten zijn onderling verschillend.

Bewijs door inductie op het aantal toppen n.

Basisgeval: Als $n = 1$, dan geldt de gestelde eigenschap altijd, omdat bomen met maar één top altijd isomorf zijn.

Inductiehypothese: Stel dat dit geldt voor twee treaps die aan (1) en (2) voldoen en $m < n$ toppen bevatten.

Te bewijzen: Geldt voor twee treaps die aan (1) en (2) voldoen en n toppen bevatten.

Neem treaps T_1 en T_2 met n toppen. Door (1) hebben ze dezelfde top met maximale prioriteit. Door (2) is er maar één dergelijke top, en door de heap-eigenschap van een treap is die top de wortel.

Neem $L(T_1)$ en $L(T_2)$, de linkse subbomen van respectievelijk T_1 en T_2. $L(T_1)$ en $L(T_2)$ bevatten sleutels die kleiner zijn dan de wortel en een kleinere prioriteit hebben dan de wortel. Omdat de wortel dezelfde is, en door eigenschappen (1) en (2), bevatten $L(T_1)$ en $L(T_2)$ dus dezelfde toppen.

Nu, aangezien $|L(T_1)|$ en $|L(T_2)|$ ten hoogste $n - 1$ kan zijn, zijn linkse subbomen door inductiehypothese isomorf. Analoog geldt dit ook voor de rechtse subbomen van T_1 en T_2.

Waarmee de gestelde eigenschap bewezen is.

Vraag 5. *Beschrijf de opzoekingsoperatie van je MyFrequencyTreap en bereken de tijdscomplexiteit. Beschrijf concreet de regel die je gebruikt om de prioriteit van frequenter bezochte toppen te verhogen.*

Prioriteitsverhoging bij `MyFrequencyTreap` gebeurt door bij elk bezoek de prioriteit van de top te verhogen met $10 * \#\text{bezoeken}$. Om de totale tijdscomplexiteit te berekenen, bepaal ik de complexiteit van elke stap van een opzoekbewerking. Er zijn twee mogelijkheden.

Gezochte top zit niet in de bom

Dit geval is simpel, we moeten enkel binair opzoeken. Aangezien de diepte van een `Treap` logaritmisch blijft, heeft binair opzoeken een tijdscomplexiteit van $O(\log n)$.

Gezochte top zit in de bom

In dit geval gebeurt de opzoekbewerking in de volgende toestappen:

1. Zoek sleutel binair, tijdscomplexiteit $O(\log n)$.
2. Verhoog het aantal bezoeken van de top, tijdscomplexiteit $O(1)$.
3. Verhoog de prioriteit volgens bovenstaande formule, tijdscomplexiteit $O(1)$.
4. Herstel de heap-eigenschap door rotaties toe te passen. De complexiteit vereist voor deze stap is ten hoogste de diepte van de treap, aangezien wij verwachten dat de diepte logaritmisch bleef, is de tijdscomplexiteit van deze stap $O(\log n)$.

Wat ons een totale tijdscomplexiteit van $O(\log n)$ oplevert.

Vraag 6. *Bepaal, bewijs en vergelijk de geamortiseerde complexiteit van een bewerking in een reeks van n bewerkingen op een initieel lege LinearFrequencyTreap en een initieel lege MyFrequencyTreap.*

Ik zou dit aantonen met de accounting methode, waarbij m het aantal toppen is van `LineairFrequencyTreap`.

	echte kost	toegekende kost
toevoegen	aantal k bezochte toppen	$2 \log m$
verwijderen	aantal k bezochte toppen	$\log m$
opzoeken	aantal k bezochte toppen	$2 \log m$

Top toevoegen

In het slechtste geval komt de top in een blad terecht en krijgt een prioriteit die hoger is dan die van de wortel, waardoor deze naar de wortel geroteerd moet worden. Toevoegen kost dus $\log m$ en het roteren naar de wortel kost ook nog $\log m$, wat nog steeds door de toegekende kost van $2 \log m$ wordt gedekt.

Top verwijderen

Er zijn twee mogelijkheden voor het slechtste geval. Ten eerste, wanneer de top in een blad zit, opzoeken wordt gedekt door $\log m$. Ten tweede, wanneer de top op een willekeurige positie zit, maar nog naar de diepte van de boom geroteerd moet worden. Dan is een fractie a van $\log m$ nodig voor het opzoeken en $(1 - a) \log m$ voor het roteren, samen nog steeds $\log m$.

Top opzoeken

Het slechtste geval treedt op wanneer de top in een blad zit en bij opzoeken met c incrementeert, terwijl alle andere prioriteiten kleiner zijn dan c. De kost voor opzoeken is $\log m$, en het roteren na het incrementeren kost ook $\log m$. Totaal nog steeds gedekt door $2 \log m$.

Zo krijgen we een bovengrens voor een bewerking: $2 \log m + \log m + 2 \log m = 5 \log m$. Voor een reeks van n bewerkingen op een initieel lege `LineairFrequencyTreap` krijgen we $O(n \log m)$. Omdat altijd $n \geq m$, geldt $O(n \log m) \leq O(n \log n)$, wat een geamortiseerde complexiteit van $O(n \log n)$ en gemiddeld $O(\log n)$ per bewerking oplevert.

Bij `MyFrequencyTreap` is het enige verschil dat het increment niet meer constant is, maar dezelfde toegekende kosten dekt nog steeds de slechtste gevallen. `MyFrequencyTreap` en `LineairFrequencyTreap` hebben dus dezelfde geamortiseerde complexiteit.

Vraag 7. *Bepaal, bewijs en vergelijk de geamortiseerde complexiteit van een bewerking in een reeks van n bewerkingen op een initieel lege MyTreap in een algemene scenario en in het ideale gebruikersscenario.*

Algemene scenario

Nu de opzoekbewerking de prioriteit zo aanpast dat we een semi-splaybewerking krijgen, kunnen we **Stelling 10** van de cursus gebruiken, met enkele aanpassingen aan de deelresultaten.

Deelresultaat 1

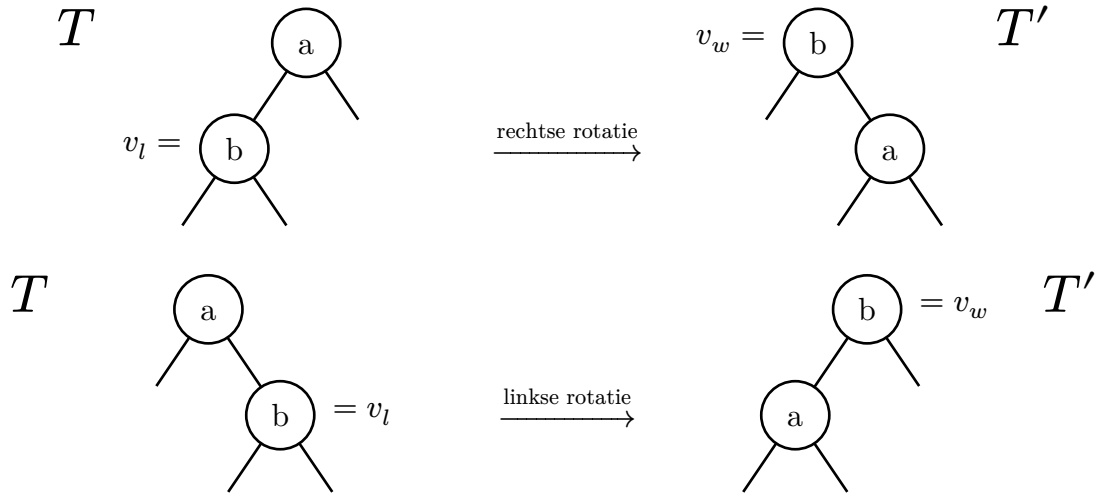
Dit blijft gelden, omdat de verwijderbewerking van `MyTreap` dezelfde is als die van `Treap`, en bij het verwijderen van een blad of een top met één kind geen rotaties nodig zijn.

Deelresultaat 2

Net zoals bij deelresultaat 1 is de toevoegbewerking dezelfde als bij `Treap`, en zolang we geen rotaties uitvoeren, blijft dit resultaat geldig.

Deelresultaat 3 (Treap)

Hier behandel ik enkel de toevoeg- en verwijderbewerkingen, aangezien voor opzoekbewerkingen de oorspronkelijke deelresultaten blijven gelden. In de opstelling van de cursus definieer ik dan v_l en v_w zoals aangegeven in **Figuur 3**.



Figuur 3: Mogelijke deelboomvervangingen bij Treap.

Wat blijft gelden is dat $A_{T'}(b) = A_T(a) \Rightarrow L_{T'}(b) = L_T(a)$. Voor potentiaal krijgen wij dus

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

Deelresultaat 4 (Treap)

Als tijdens een toevoegbewerking of opzoekbewerking in een boom T langs een pad met $2s+1$ toppen deelbomen vervangen worden, dan vervangen wij ten hoogste $2s$ keer de deelboom. De definitie $v_{l,i}$, $v_{w,i}$ en T_i is dezelfde als in cursus. Wij krijgen dus

$$\Phi(T') = \Phi(T) + \sum_{i=1}^{2s}(\Phi(T_i) - \Phi(T_{i-1})) \leq \Phi(T) + \sum_{i=1}^{2s}\Big(L_{T_i}\big(v_{w,i}\big) - L_{T_{i-1}}\big(v_{l,i}\big)\Big)$$

Hier ook, zoals aangetoond in **Figuur 3**, is de nieuwe wortel van een net geplaatste deelboom de laagste top in de volgende stap, dus $v_{w,i} = v_{l,i+1}$. Zo krijgen wij

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

Net zoals in de cursus definieer ik $\overline{T}$ als de boom bij een toevoeg- of verwijderbewerking na het toevoegen of verwijderen, maar vóór de rotaties. En $k \leq 2s+3$ net zoals in deelresultaat 4.

	echte kost	gewijzigde kost
toevoegen	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ $= k + \Phi(T') - \Phi\big(\overline{T}\big) + \Phi\big(\overline{T}\big) - \Phi(T)$ nu wordt deelresultaat 2 gebruikt: $\leq k + \log\lvert\overline{T}\rvert + \Phi(T') - \Phi\big(\overline{T}\big)$ $\lvert\overline{T}\rvert = \lvert T'\rvert$ en deelresultaat 4 (treap): $\leq k + \log\lvert T'\rvert + \log\lvert T'\rvert \leq 2\log\lvert T'\rvert$
verwijderen	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ $= k + \Phi(T') - \Phi\big(\overline{T}\big) + \Phi\big(\overline{T}\big) - \Phi(T)$ nu wordt deelresultaat 1 gebruikt: $\leq k + \Phi(T') - \Phi\big(\overline{T}\big)$ nu wordt deelresultaat 4 (treap) gebruikt: $\leq k + \log\lvert T'\rvert \leq \log\lvert T'\rvert$
opzoeken	aantal k bezochte toppen	$k + \Phi(T') - \Phi(T)$ nu wordt orspronkelijke deelresultaat 4 gebruikt: $\leq k + 2\log\lvert T'\rvert - 2s \leq 2\log\lvert T'\rvert + 2$

Zo krijgen we voor $n > 1$ bewerkingen in het algemene geval een geamortiseerde complexiteit van $O(n \log n)$. De gewijzigde kost per bewerking is $O(\log n)$ en de potentiaal aan het einde is minstens even groot als in het begin.

Ideale gebruikersscenario

Zoals besproken in **Hoofdstuk 2.5** is het ideale scenario dat we eerst een aantal toppen toevoegen en vervolgens dezelfde top meerdere keren opzoeken. De beperkende factor hierbij zijn de toevoegbewerkingen, omdat die volgens het `Treap`-algoritme verlopen. Zoals besproken in **Vraag 6** kost het uitvoeren van m toevoegingen $O(m \log m)$.

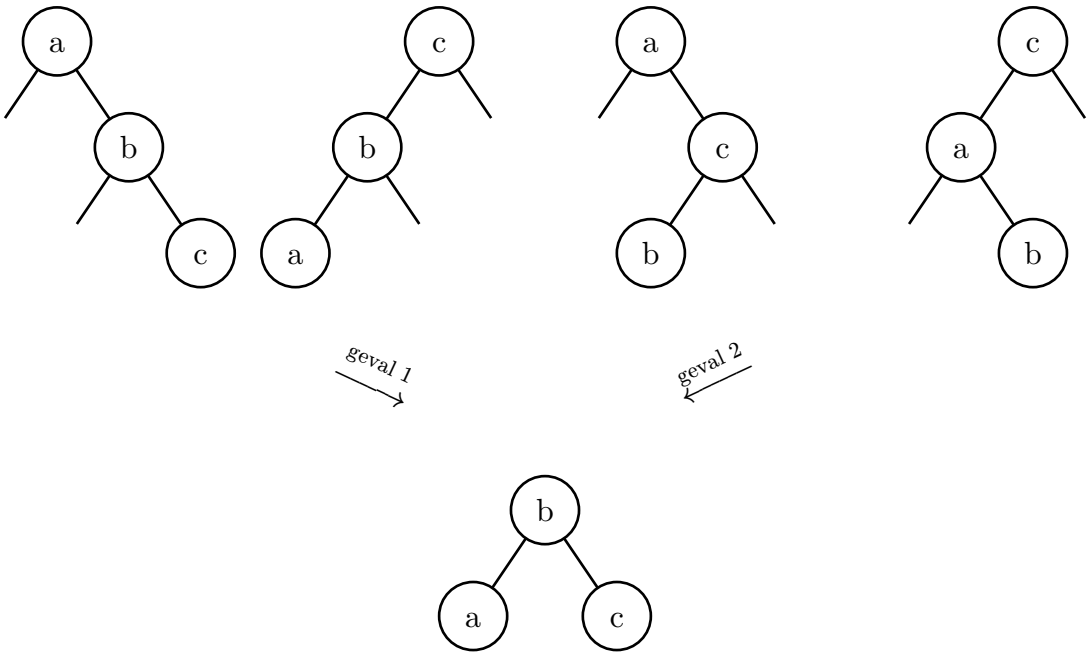
Als er oorspronkelijk s toppen op het pad van de wortel naar de gezochte top liggen, dan is de diepte na één semi-splay bewerking $\lceil \frac{s}{2} \rceil$ zoals vermeld in cursus pagina 16. Na t keer opzoeken is de diepte dus $\lceil \frac{s}{2^t} \rceil$. De top komt in de wortel (of een kind ervan) wanneer $2^t \geq s$, wat neerkomt op $t \geq \log(s)$ opzoeken.

Zo zien wij dat in zowel ideale gebruikersscenario als in het algemene geval de geamortiseerde complexiteit voor $n > 1$ bewerkingen op een initieel lege `MyTreap` $O(n \log n)$ is, met dus gemiddeld $O(\log n)$ per bewerking.

2 Implementatie

2.1 SemiSplayTree

Implementatie van elke functie van `SemiSplayTree` ziet er bijna hetzelfde uit: voer de gevraagde bewerking uit zoals in een binaire boom en roep vervolgens de functie `semiSplay()`. Ik zal dus enkel de functie bespreken die cruciaal is voor de uitvoering van `semiSplay()`, namelijk `restruct()`. In eerste instantie heb ik die geïmplementeerd door gewoon alle 4 mogelijke gevallen van figuur 5 uit de cursus te definiëren, maar na implementatie van `Treap` had ik gezien dat dit eigenlijk geen 4 maar 2 gevallen zijn. Het eerste zijnde dat we een *rechte* lijn van 3 toppen hebben en het andere dat we een *zigzag* lijn hebben. Hierdoor is het mogelijk om een zeer korte implementatie van `restruct` te schrijven die als volgt werkt.



Figuur 4: Twee gevallen van de semi-splay operatie.

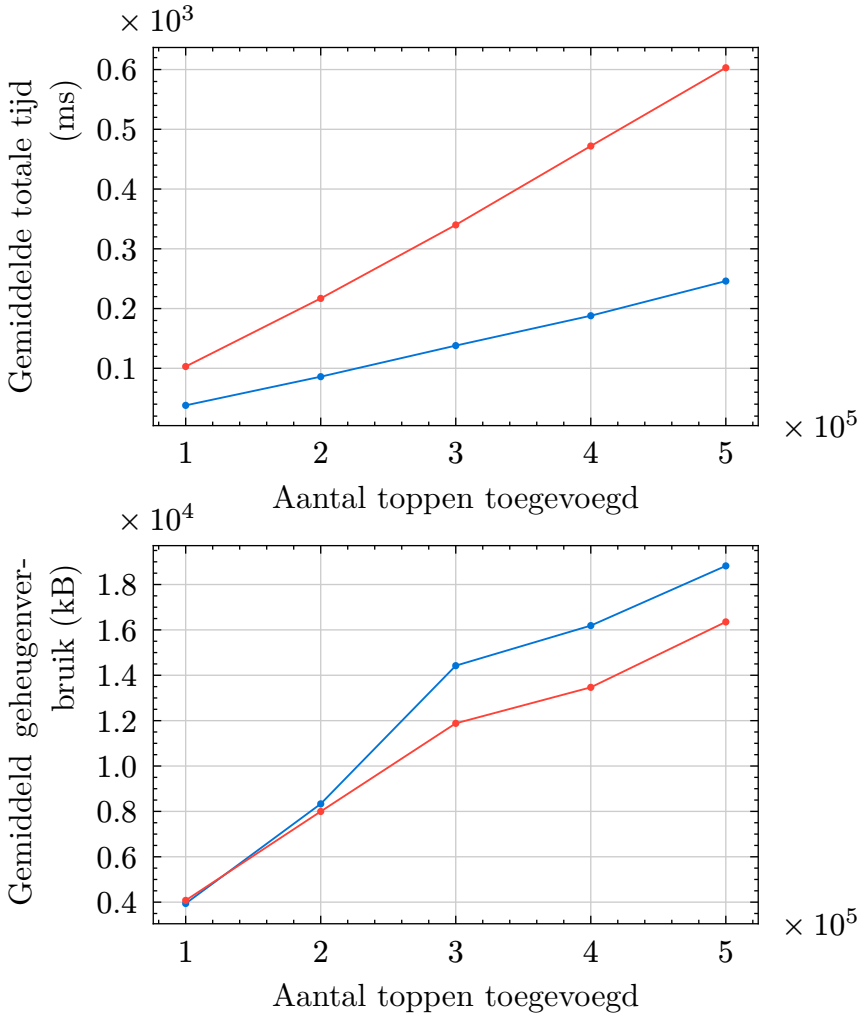
Voor de uitleg gebruik ik de namen van de toppen zoals gegeven in **Figuur 4**. In het eerste geval is het voldoende om één rotatie toe te passen op b en zijn ouder om de semi-splay bewerking op de drie betrokken toppen uit te voeren. Wat ik met rotatie bedoel, staat in **Figuur 6**.

In het tweede geval zijn er twee rotaties nodig: twee keer op b en zijn ouder, zodat b de nieuwe wortel van deze drie toppen wordt. Hiermee krijg je een eenvoudige implementatie en kan je dezelfde rotatiemethodes gebruiken als in zowel `SemiSplayTree` als in de `Treap`-varianten.

Dit bleek niet alleen handig voor de implementatie, maar ook voor het bewijs van **Vraag 7**.

2.1.1 Optimalisaties

In dit deel bespreek ik waarom de initiële implementatie van semi-splay, waarbij het pad werd bijgehouden in een stapel, is vervangen door het toevoegen van een ouderveld aan elke top. De motivatie hiervoor was dat bij het bekijken van de profilerresultaten de bewerkingen `push()` en `pop()` de grootste hoeveelheid tijd in beslag namen.



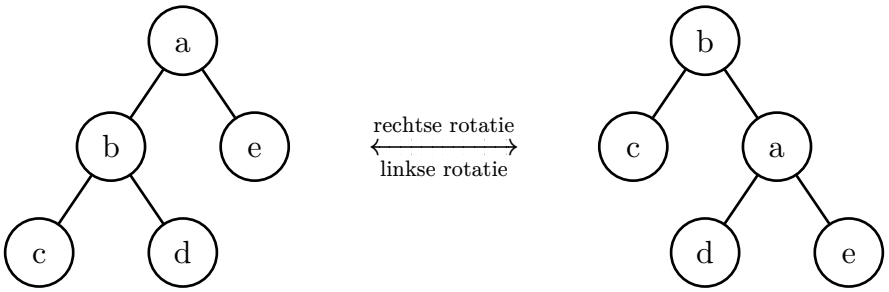
Figuur 5: Semi-splay met bijhouden van ouder in blauw, semi-splay met stapel in rood.

De conclusie van deze benchmark is dat de strategie met bijhouden van ouder veel tijdsefficiënter is dan de stapel, zoals aangetoond in de bovenste grafiek van **Figuur 5**, terwijl de stapel iets meer geheugenefficiëntie biedt wanneer de boom voldoende groot is (meer dan 200.000 toppen), zoals aangetoond in de onderste grafiek van **Figuur 5**. Aangezien de verbetering in uitvoeringssnelheid een veel groter effect heeft dan de verbetering in geheugengebruik, is ervoor gekozen om de implementatie met bijhouden van ouder te gebruiken.

2.2 Treap

Aangezien het enige wat in `Treap` verschilt van een binaire boom de toevoeg- en verwijderbewerking is, zou ik ook enkel de implementatie hiervan bespreken.

Bij beide bewerkingen spelen rotaties de grootste rol, namelijk rechtse rotatie en linkse rotatie, die tegengesteld aan elkaar zijn. Hoe die gebeuren, kun je zien in **Figuur 6**.



Figuur 6: Linkse en rechtse rotatie

Daarbij had ik de hulpfunctie `rotateUpWhileNotHeap()` gedefinieerd. Deze functie voert rotaties uit indien de prioriteit van de top groter is dan die van zijn ouder. Als onze top b is en de ouder a zoals in **Figuur 6**, wordt een rechtse rotatie uitgevoerd, en als het omgekeerd is, onze top a en de ouder b, wordt een linkse rotatie uitgevoerd. Dit proces blijft doorgaan totdat de ouder van de top een prioriteit heeft die groter of gelijk is, of totdat de top de wortel van de boom is.

Toevoegen

Eerst voegen wij deze top binair toe aan de boom. Indien de sleutel al in de boom zit, stoppen wij. Anders kennen wij een willekeurige prioriteit toe aan de nieuwe top. Meer over hoe ik dit doe, leg ik uit in **Hoofdstuk 2.2.1**. Vervolgens geven wij de top door aan `rotateUpWhileNotHeap()`.

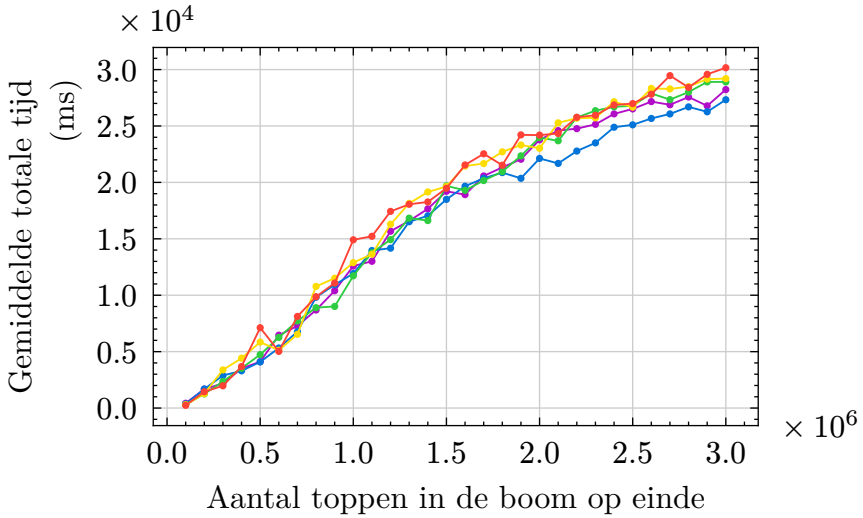
Verwijderen

Eerst zoeken wij de top die wij willen verwijderen binair in de boom. Als onze top daarbij twee kinderen heeft, voeren wij een rotatie uit, zodat het kind met de grootste prioriteit boven komt. Dit blijven wij herhalen totdat onze top in een blad zit of slechts één kind heeft. Dan verwijderen wij deze gewoon of vervangen wij deze door het enige kind.

2.2.1 Optimalisaties

Hier zou ik bespreken hoe wij een willekeurige prioriteit kiezen voor een nieuw toegevoegde top. Aangezien het duidelijk is dat wij geen vaste seed gaan gebruiken, is de enige parameter die wij kunnen aanpassen, de bovengrens bij het genereren van de prioriteit.

Logisch is dat hoe groter onze bovengrens is, hoe minder kans dat wij toppen met dezelfde prioriteit zullen hebben. De vraag is enkel of dit iets is wat wij willen. De eenvoudigste weg om te zien hoe het veranderen van de bovengrens invloed heeft op de efficiëntie, is het uitvoeren van een benchmark. Hierbij voeren wij een benchmark uit van het gemiddelde geval waarin wij aantal toppen toevoegen, aantal toppen zoeken, aantal toppen verwijderen en dan die stappen herhalen.



Figuur 7: Bovengrens 10.000 in paars, 100.000 in blauw, 1.000.000 in groen, 10.000.000 in geel, `Integer.MAX_VALUE` in rood.

Op basis van **Figuur 7** kunnen wij concluderen dat 100.000 iets beter zal presteren naarmate het aantal toppen toeneemt.

Achteraf heb ik bij het schrijven van het verslag meer tijd besteed aan literatuur over `Treap`'s. Botsingen willen we zo veel mogelijk vermijden, omdat de berekening van de verwachte diepte uitgaat van unieke prioriteiten. Bij veel gelijke prioriteiten geldt de theoretische logaritmische diepte niet meer, maar de boom presteert daardoor niet direct slecht.

Ik had de benchmarks met een bovengrens voor prioriteiten beter kunnen weglaten, maar ik geef er de voorkeur aan mijn fout te erkennen in plaats van deze te verbergen.

2.3 LineairFrequencyTreap

Het enige waarin `LineairFrequencyTreap` van `Treap` verschilt, is de opzoekoperatie, dus ik bespreek alleen dit.

Bij de opzoekoperatie wordt, als de sleutel in de boom zit, de prioriteit van de top met die sleutel verhoogd met een constante. Vervolgens roteren we, net zoals bij `Treap`, de top omhoog totdat de heap-eigenschap hersteld is.

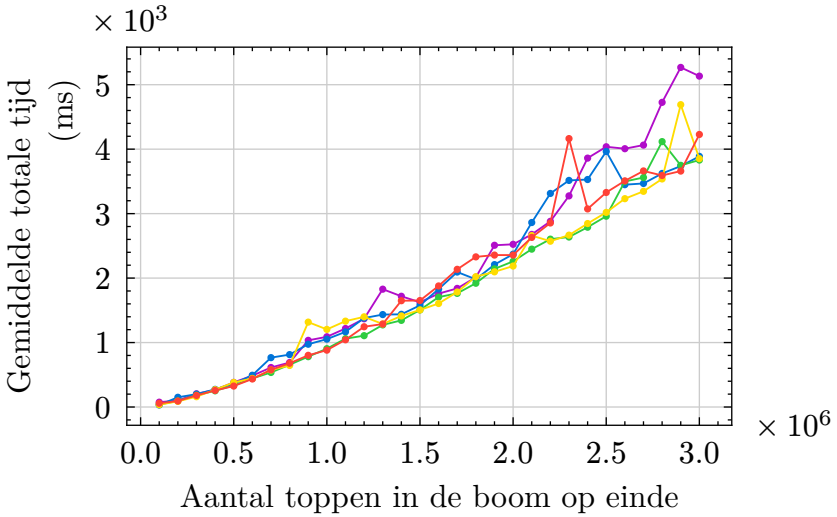
Hierbij moet de constante groot genoeg zijn zodat, naarmate het aantal bezoeken stijgt, de prioriteit ook aanzienlijk toeneemt. Ik heb gekozen voor 100 met een bovengrens van 10.000.

2.4 MyFrequencyTreap

Bezoek operatie gebeurt hier net zoals bij `LineairFrequencyTreap`. De prioriteit verhoging functie is veel anders, die kan je terugvinden in **Vraag 5**. Perfecte scenario waarvoor zo een prioriteit verhoging zinvol is, is indien wij dezelfde top vaak gaan opzoeken. Zo krijgen wij sneller prioriteitgroei, wat resulteert in dat top veel sneller dichter bij de wortel komt en er dus minder toppen te bezoeken zijn om die dan opnieuw te zoeken.

2.4.1 Optimalisaties

Net zoals bij `Treap` gaan had ik benchmarks uitgevoerd om beste bovengrens voor genereren van prioriteit te kiezen.

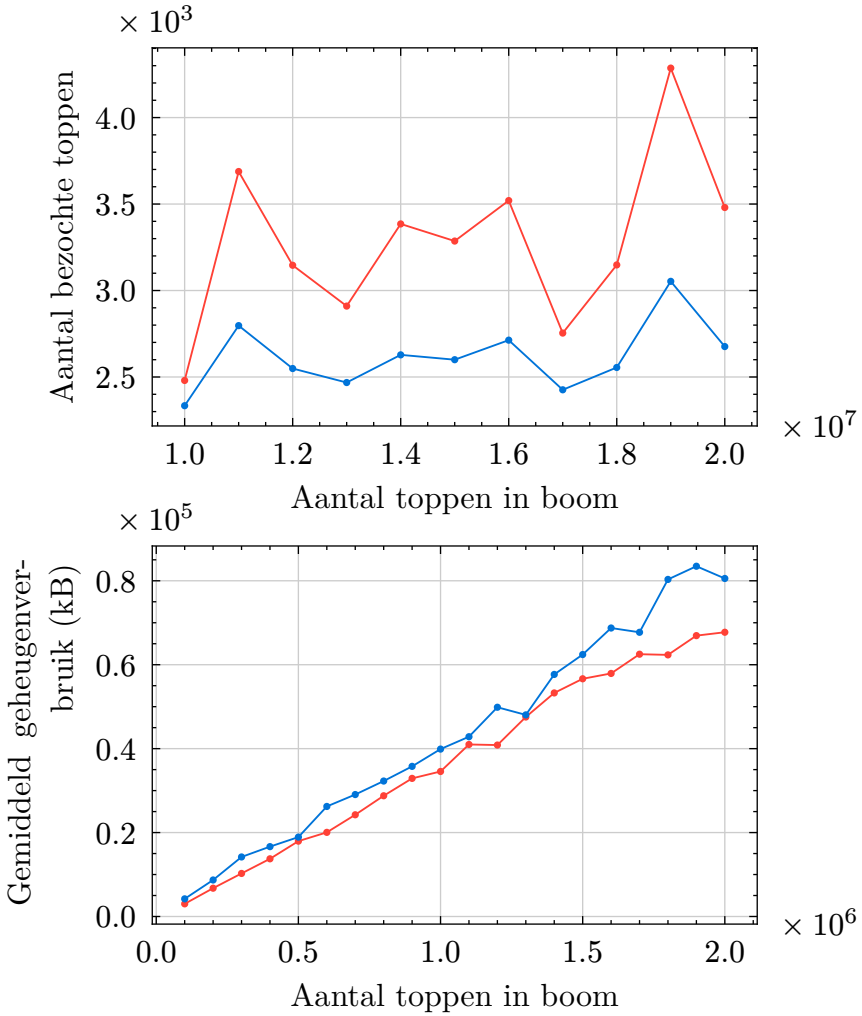


Figuur 8: Bovengrens 10.000 in paars, 100.000 in blauw, 1.000.000 in groen, 10.000.000 in geel, `Integer.MAX_VALUE` in rood.

Uit de resultaten van **Figuur 8** is 100.000 gekozen, laag genoeg voor merkbare prioriteitstoename en vergelijkbare prestaties als andere grenzen. Zelfde opmerking geldt als bij **Hoofdstuk 2.2.1**.

2.4.2 Vergelijking LineairFrequencyTreap en MyFrequencyTreap

Voor het perfecte scenario van `MyFrequencyTreap` verwachten we dat als we dezelfde top vaak opzoeken, er minder totale bezochte toppen zullen zijn dan bij `LineairFrequencyTreap`. Dit is ook exact wat we zien in de benchmarkresultaten van het aantal bezoeken. `MyFrequencyTreap` gebruikt daarbij meer geheugen, omdat deze ook het aantal bezoeken van een top moet opslaan. Die presteert natuurlijk alleen beter als beide bomen dezelfde grens voor de generatie van prioriteit hebben. In deze benchmark was die in beide bomen 10.000.



Figuur 9: LineairFrequencyTreap in rood, MyFrequencyTreap in blauw.

2.5 MyTreap

Stel dat wij een boom willen die zich veel sneller aanpast dan `LineairFrequencyTreap` en `MyFrequencyTreap`, dat dee me direct denken aan een `SemiSplayTree`. Maar alhoewel dit grootste sterkte is van `SemiSplayTree` resulteert die in een boom die in gemiddelde geval toch vrij slecht presteert omdat elke operatie een sterke invloed op structuur van de boom heeft.

Nu wat als wij toch die sterkte willen gebruiken en nog steeds relatief goede performantie in gemiddelde geval hebben. Dan willen wij een `Treap` die bij zoek operatie zo prioriteit aanpast dat wij een `semi-splay` bewerking krijgen. Dit is exact wat ik bij `MyTreap` had gedaan.

Om de in **Figuur 4** getoonde operatie uit te voeren, veranderen we de prioriteit van b als

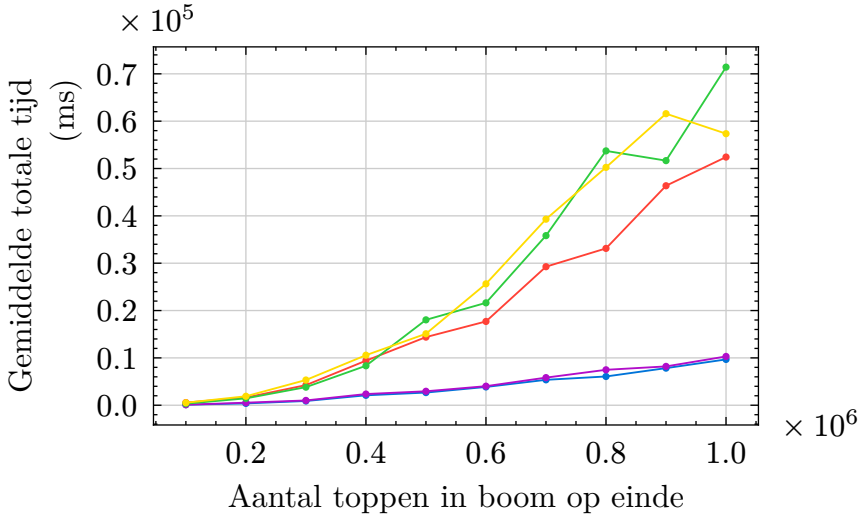
$$\text{b.prioriteit} = \text{prioriteit van bovenste van drie toppen} + 1$$

Daarbij noem ik de bovenste toppen in de vier bomen respectievelijk a, c, a en c. Na de prioriteitsaanpassing is het voldoende om b aan `rotateUpWhileNotHeap()` door te geven om de semi-splay operatie voor die drie toppen te verkrijgen.

2.5.1 Vergelijking van MyTreap en andere bomen

Perfecte geval voor MyTreap

Voor het perfecte scenario van MyTreap kiezen we een aantal bladeren en zoeken we elke blad diepte(blad) + c keer op. De bedoeling is dat na diepte aantal keer opzoeken een top bij `MyTreap` en `SemiSplayTree` in de buurt van de wortel en volgende opzoeken O(1) zouden vereisen.



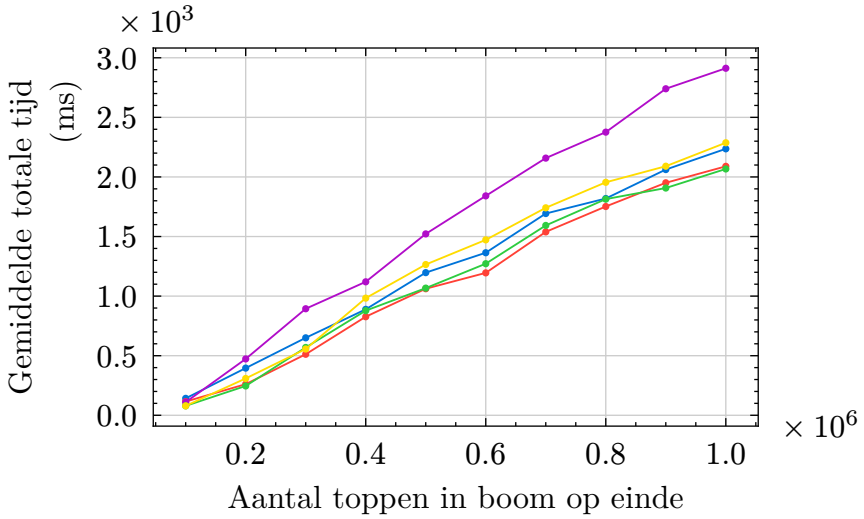
Figuur 10: Ideale scenario. MyTreap in blauw, Treap in rood, SemiSplayTree in paars, LineairFrequencyTreap in groen en MyFrequencyTreap in geel.

Wat mij verraste is dat in dit geval `LineairFrequencyTreap` en `MyFrequencyTreap` slechter presteren dan Treap. Dit komt waarschijnlijk doordat we veel te weinig keer opzoeken waardoor in beide frequentietreaps de prioriteit van de opgezochte top niet genoeg wordt verhoogd. Daardoor moeten we blijven roteren terwijl de top laag in de boom blijft staan.

Concluderend krijgen wij gewenste resultaat bij perfecte geval van `MyTreap` waar `MyTreap` even goed als `SemiSplayTree` presteert.

Gemiddelde geval

Hier benchmarken we het eerder besproken gemiddelde scenario. Ook hier presteren alle `Treap`-varianten, inclusief `MyTreap` dus, beter dan `SemiSplayTree`.



Figuur 11: Gemiddelde scenario. MyTreap in blauw, Treap in rood, SemiSplayTree in paars, LineairFrequencyTreap in groen en MyFrequencyTreap in geel.