Project Algoritmen en Datastructuren 2

De semi-splay-2-3-boom

Verslag door Thibaud Collyn

1) Theoretische vragen

Vraag 1

/

Vraag 2

Het maximale aantal toppen in het splaypad met maar 1 key (*MaxSingleKey ~ MSK*) kan gevonden worden door:

$$MSK = \lceil (k-1)/2 \rceil = \lceil k/2 \rceil - 1$$

Dit volgt uit het feit dat er per splay bewerking maximaal 1 top met maar 1 key kan voortkomen (dit volgt uit de gegeven deelbomen). Het enige wat we nu nog moeten weten is het maximale aantal splay bewerkingen in een boom van k toppen. Dit is echter gewoon gelijk aan $\lceil n/2 \rceil$ waarbij n het aantal bogen in de boom is en aangezien het aantal bogen gelijk is aan het aantal toppen min 1 is de formule bewezen. ■

1) Implementaties van de zoekbomen

Normale 2-3-boom

Bij de implementatie van de normale 2-3-boom overloop ik, bij toevoegen als verwijderen, alle mogelijke niet gebalanceerde subtrees die zich na de operatie kunnen voordoen om vervolgens de boom weer in balans te brengen. Dit gebeurt allemaal door recursieve add en remove balanceer functies.
Ik heb ook geprobeerd om zo min mogelijk hulp objecten aan te maken in de recursieve functies om de performantie niet te hard naar beneden te halen.

Opmerking:
Bij mijn implementatie van de 2-3-boom ben ik bij de remove operatie vergeten check of de key die verwijderd wordt al dan niet in de boom zit, dit zorgt dus voor een crash als je een key probeert te verwijderen die niet in de boom zit. Aangezien dit een eerder kleine fout is die met een simpele if conditie op te lossen valt vermeld ik het hier (na de aanpassing te maken in mijn code werkt de boom perfect).

Semi-splay bomen

Algemeen

Bij de bottom-up en top-down semi-splay bomen zijn er veel dingen die gedeeld worden aangezien de top-down boom van de bottom-up boom over erft, daarom zal ik hier wat algemene implementatiedetails bespreken.

In beide gevallen maak ik gebruik van een algemene splay methode die gebruik maakt van een stack om te splayen. Het concept is hier dat de stack het splaypad bevat die de juist uitgevoerde bewerking(add, remove of contains) heeft genomen. Hierna zal de methode recursief de volgende drie nodes in het splaypad herbalanceren tot het einde van het splaypad bereikt is.
De implementatie van de algemene splay methode ziet er nogal omslachtig uit aangezien deze bijna elk geval van subtree appart overloopt. Desondanks heb ik wel geprobeerd om zo min mogelijk code duplicatie te hebben en zoveel mogelijk algemene stappen tussen cases te bundelen.

Een nadeel aan deze implementatie van de semi-splay-2-3-boom is dat er meer nodes met maar 1 key zullen zijn, aangezien het goed kan zijn dat na het toevoegen van een node met maar 1 key die naar omhoog gesplayed wordt en 2 kinderen zal hebben. D.w.z. dat we hier geen tweede key meer gemakkelijk aan kunnen toevoegen om een node met 2 keys te maken aangezien dit problemen kan veroorzaken met de kinderen van de node.

Bottom-up

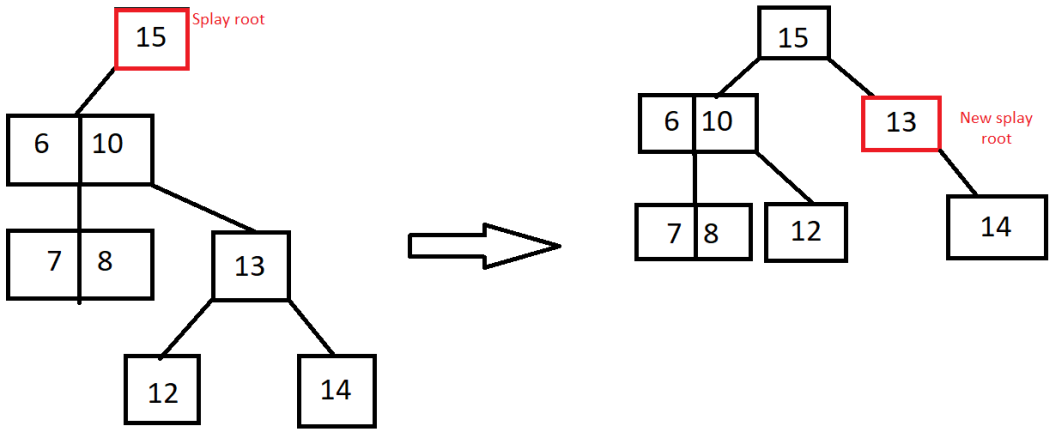
In de bottom-up implementatie wordt het splaypad altijd opnieuw gezocht door een apparte functie.
Er wordt ook na elke opperatie gesplayed tot de root.

Top-Down

Het grootste verschil bij deze implementatie is dat het splaypad appart word bijgehouden in een veld en dat het na elke splaybewerking wordt aangepast.
Hier is het belangerijkste verschil dat de contains methode herschreven is zodat deze top down werkt en de add en remove aangepast zodat deze gebruik maken van de top-down contains.

Optimalisaties

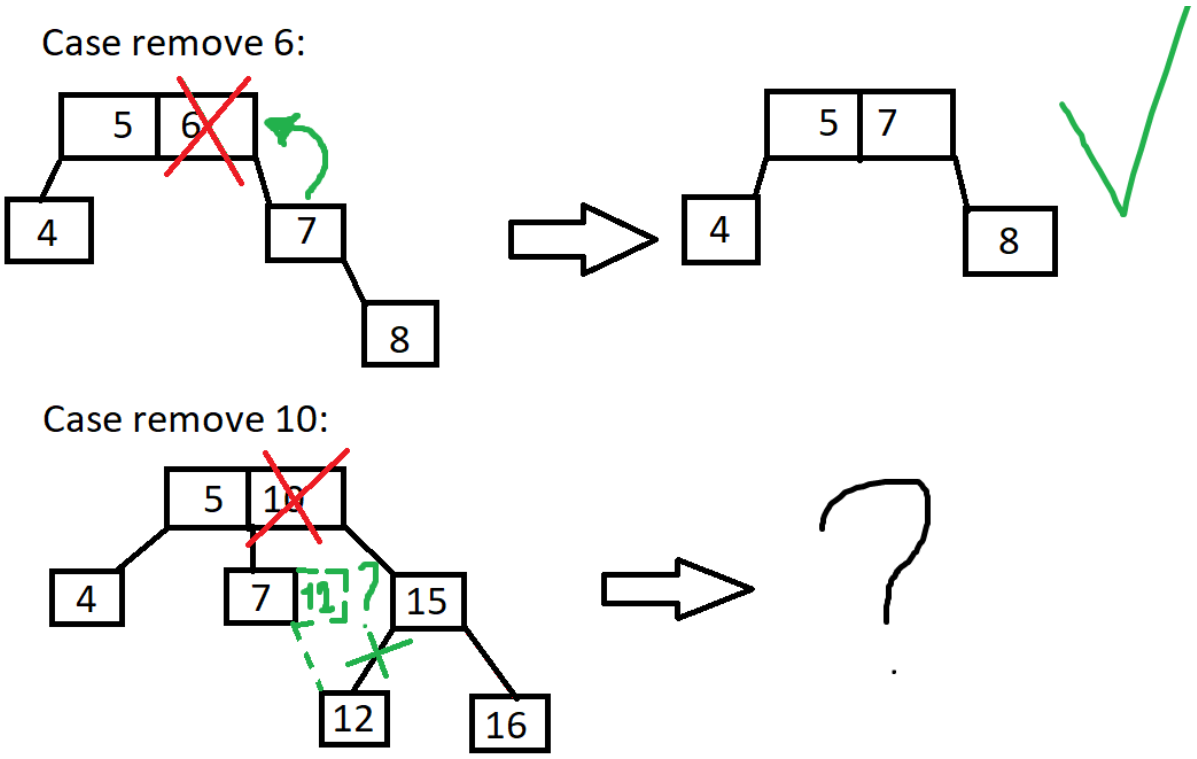
Een eerste optimalisatie die ik zou doorvoeren is de manier waarop het splaypad bekomen wordt. Ik weet dat het constant opnieuw initialiseren van een stack(in de bottom-up) en het toevoegen en verwijderen van elementen hierin toch wel wat overhead met zich mee zullen brengen. Daarom zou ik bij een andere implementatie werken met een zo gezegde 'splay-root'. Dit zou gewoon een veld zijn die een enkele node bijhoud, zo kan de algemene splay methode altijd met .getParentNode() methodes werken.
Bij de bottom-up implementatie zou de splay root dan altijd de effectieve root van de boom zijn en bij de top-down implementatie zou deze root na elke splay-stap van 3 nodes naar beneden bewegen.
Voorbeeld top-down opzoeken van key met splay-root:



Als men 14 zou opzoeken zouden we in een top-down implementatie al een keer willen splayen nadat de node die 13 bevat bereikt is, dit doen we tot we aan de splay-root zitten. Hierna veranderen we de splay root naar de node die 13 bevat als we hypothetisch gezien nog 2 nodes verder gaan zullen we opnieuw splayen maar dit maal niet tot de effectieve root 15 maar tot de splay root 13

Een tweede optimalisatie zou zijn om bij bepaalde splay en remove gevallen i.p.v. direct te splayen eerst kijken of het mogelijk is om een node met 2 keys te maken. Dit zou moeten helpen tegen lange takken in de boom die waardoor er minder toppen bezocht moeten worden.
Deze optimalisatie zou wel heel wat codeer werk zijn aangezien er veel meer extra checks zouden moeten plaatsvinden om te controleren of 2 nodes al dan niet samen gevoegd kunnen worden.

Voorbeeld:

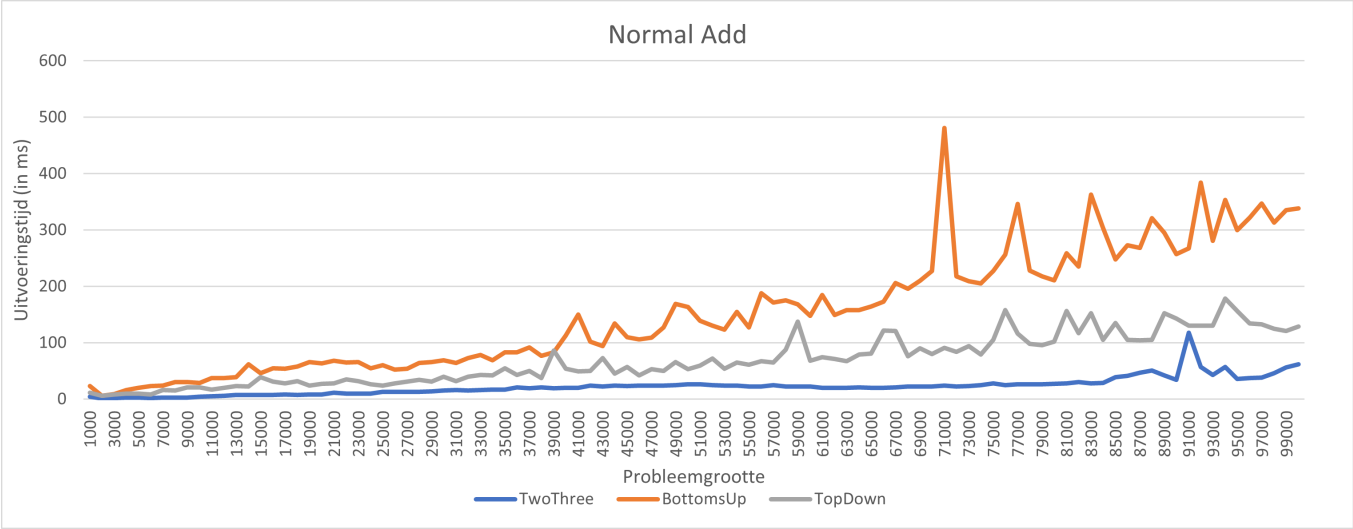


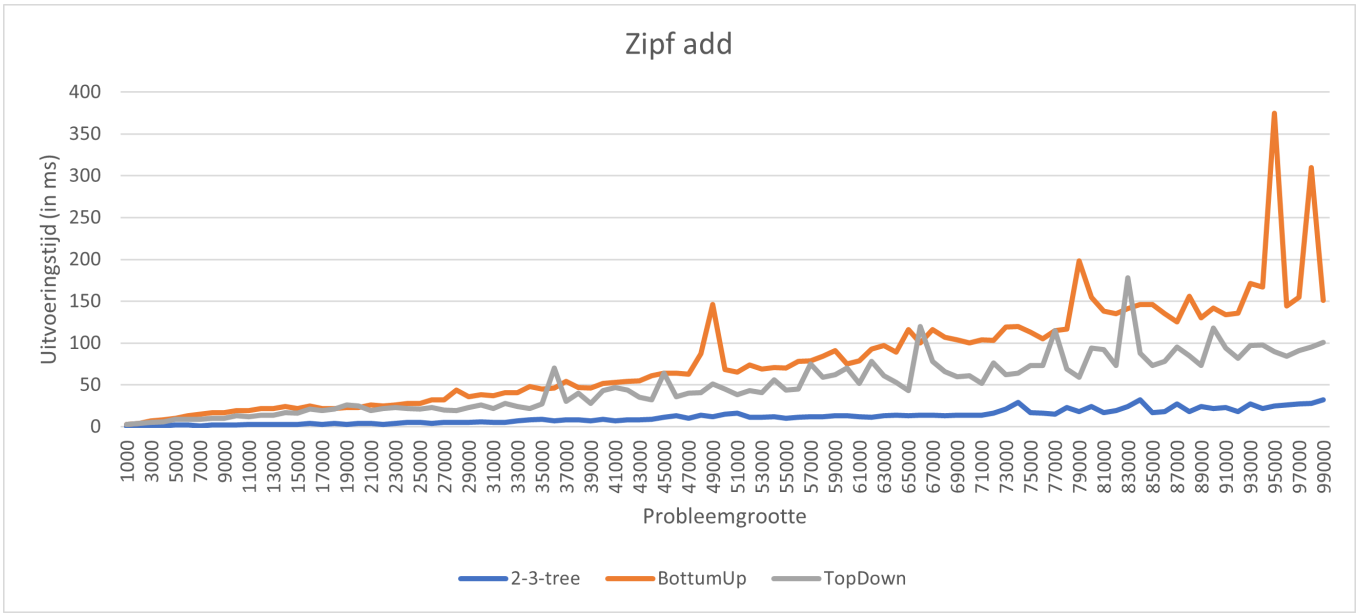
In dit voorbeeld wordt als snel duidelijk dat 2 situaties die sterk op elkaar gelijken toch niet zo voor de hand liggend zijn. In de 2de case is er namelijk ook de vraag of we 12 toevoegen aan 7 of een kind maken van 7 indien we 15 op de plaats van 10 willen zetten. Dit voorbeeld maakt duidelijk dat dit soort optimalisatie heel veel extra condities en programmeer werk met zich mee zou brengen...

1) Performantie a.d.h.v. benchmarks

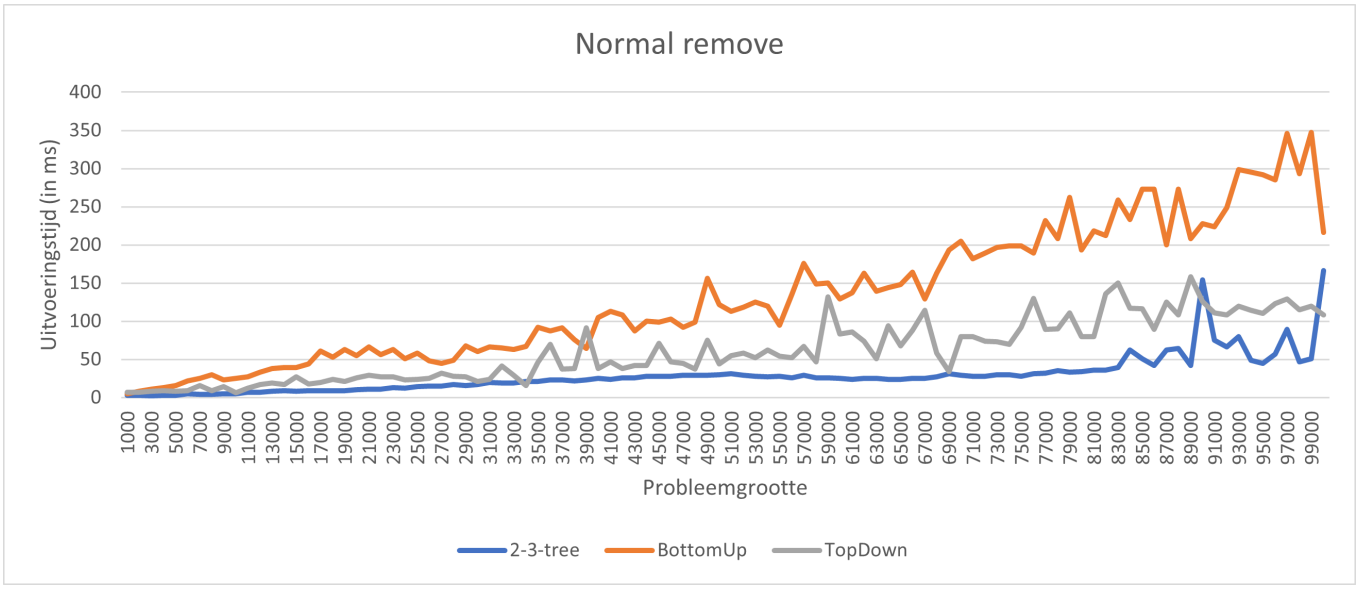
Hier zullen we de performantie van de basis operaties overlopen met 2 verschillende soorten verdelingen (normaal en zipf) van alle implementaties.

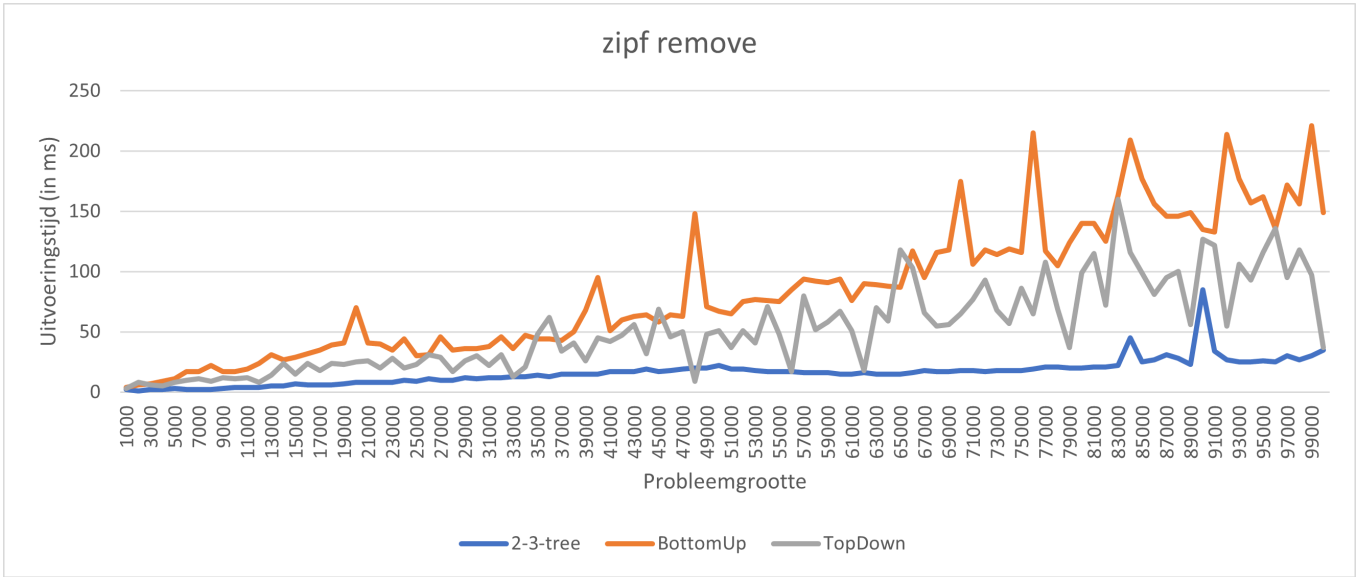
Toevoegen



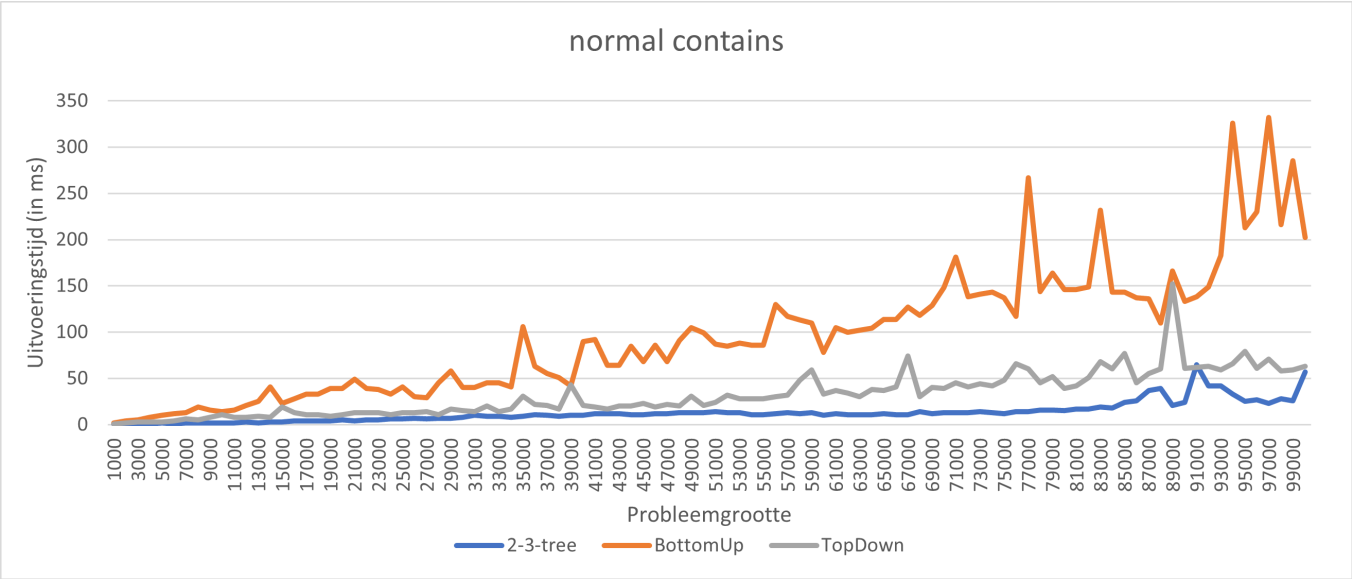


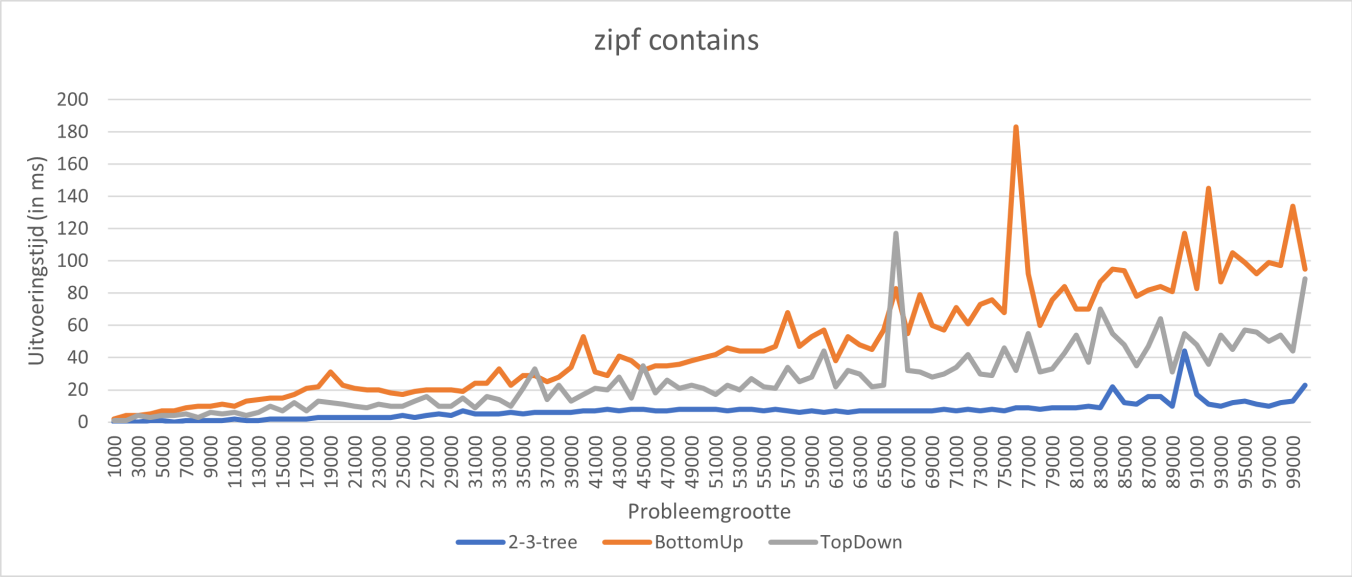
Verwijderen





Opzoeken





Conclusie bench testen

Zoals we duidelijk kunnen zien blijft de gewone 2-3-tree het meest performant en consistent qua uitvoeringstijd vergeleken met beide semi splay implementaties. Dit komt hoogst waarschijnlijk door de uitschietende takken in de semi-splay implementatie die eersder besproken zijn in de algemene implementatie details van de semi-splay-2-3-bomen.

Er is ook een duidelijk verschil tussen de bottom-up en de top-down semi-splay bomen, top-down is namelijk toch wat beter. Over het algemeen presteren de zipf verdelingen beter dan de normale verdeling.
Het is voor de hand liggend dat de top-down semi-splay implementatie beter presteerd aangezien deze niet altijd een nieuwe stack aanmaakt om het splaypad te zoeken en aangezien deze het splaypad algemeen minder moet overlopen.