Janne Cools

# De beste binaire zoekboom: verslag

## Theoretische vragen

Bij een zo goed mogelijk gebalanceerde binaire boom is het verschil in hoogte van de linker- en rechterdeelboom hoogstens 1. Hierdoor is de diepte gemiddeld O(log m) met m het aantal toppen in de boom. Om de complexiteit van n bewerkingen op de boom te kunnen bepalen, moet er gekeken worden naar het slechtste geval.

Het toevoegen van n toppen die telkens het kind van de vorig toegevoegde top worden, zal het meeste tijd vragen, aangezien herhalend hetzelfde pad zal doorlopen worden die steeds 1 top langer wordt.

De complexiteit wordt dan berekend op volgende manier (waarbij de som berekend wordt voor i met waarden van 0 tot en met n-1):

$O(\log m) + O(\log m)+1 + O(\log m)+2 + \ldots + O(\log m)+(n-1) = \sum (i + O(\log m))$
$= \sum i + n*O(\log m) = (n-1)*(n-2)/2 + n*O(\log m) = O(n^2) + n*O(\log m)$

Hierdoor is de complexiteit voor n bewerkingen $O(n^2) + n*O(\log m)$ en de complexiteit voor 1 bewerking $O(n) + O(\log m)$.

Stel dat bij deze n bewerkingen er hoogstens log n toevoegingen zijn. Als opnieuw het slechtst geval wordt nagekeken, zijn er exact log n toevoegingen. Bij het gebruiken van de bewerking in de vorige alinea, moet de "n" vervangen worden door "log n" wat resulteert in de complexiteit $O((\log n)^2) + \log n * O(\log m)$ voor log n toevoegbewerkingen dus $O(\log n) + O(\log m)$, wat vereenvoudigd kan worden naar $O(\log m*n)$ per bewerking. De overige bewerkingen, die enkel verwijderingen en opzoekingen zijn, worden uitgevoerd in een gemiddelde tijd van $O(\log n)$ per bewerking. De totale complexiteit per bewerking is dan $O(\log n + \log (m*n)) = O(\log (m*n^2))$.

Bij de zoekboom die opgebouwd wordt met het algoritme die gebruik maakt van de gewichten, zal de diepte niet altijd $O(\log n)$ zijn maar wel altijd kleiner dan n/2 met n het aantal toppen in de boom. Dit zorgt er echter wel voor dat het slechtste geval (namelijk n toppen toevoegen die telkens het kind worden van de vorig toegevoegde top) lineair is. Want als de diepte gelijk is aan m/2 met m aantal toppen in de boom, dan wordt de complexiteit:

$O(m/2) + O(m/2)+1 + \ldots + O(m/2)+(n-1) = \sum (i + O(m/2)) = \sum I + n*O(m/2) =$
$(n-1)*(n-2)/2 + n*O(m/2) = O(n^2) + n*O(m/2) = O(n^2) + n*O(m)$

Hierdoor is de complexiteit voor n bewerkingen $O(n^2) + n*O(m)$ en de complexiteit voor 1 bewerking $O(n+m)$.

## Uitleg bij implementatie bomen

Voor de boom SemiSplayTree heb ik mij gebaseerd op de uitleg in de cursus op pagina's 14 en 15. Hierbij heb ik het algoritme overgenomen samen met de methode om het pad naar de gezochte top te bepalen. In een while-lus bepaal ik telkens de 3 toppen, die nodig zijn voor de semi-splay, en de ouder van de top die het dichtst bij de wortel is. Daarna vervang ik deze deelboom door een complete binaire deelboom en ken ik alle kinderen van deze 3 toppen toe aan hun nieuwe ouder.

Vervolgens heb ik voor de implementatie van de methode "optimize" van de OptimalTree mij gebaseerd op een filmpje waar een voorbeeld wordt gegeven (de link staat als commentaar bij de code). Voor dit algoritme houd ik 2 tweedimensionale lijsten bij, een lijst die de kosten en een lijst die de wortel bijhoudt van een groep toppen. Voor elke groep toppen van een gegeven lengte bepaal ik voor elke mogelijke wortel de kost van die deelboom en plaats ik het minimum daarvan in de lijst.

Als beide lijsten aangevuld zijn, construeer ik de nieuwe boom met de hulpmethode "restructure". Hierin voeg ik recursief de wortel van de groep toppen (gegeven met de indexen start en stop) toe aan de boom als linker- of rechterkind van de ouder die meegegeven wordt als argument.

Deze implementatie heb ik ook gebruikt om de methode "optimize" met de extra parameter van externe gewichten te laten werken. De enige aanpassing is het berekenen van de variabele "weight". In plaats van enkel de som van de interne gewichten te bepalen, bereken ik de externe gewichten erbij.

Om codeduplicatie te vermijden roep ik de methode "optimize" met 3 parameters op bij de methode "optimize" met 2 parameters, waarbij ik als derde parameter een lijst meegeef waar alle externe gewichten hetzelfde zijn, namelijk nul.

Voor de laatste zelforganiserende zoekboom heb ik enkele heuristieken opgezocht en met elkaar vergeleken.

De zoekbomen die ik het vaakst tegenkwam, waren de rood-zwart boom, 2-3 boom en AVL-boom. De rood-zwart boom en 2-3 boom vond ik minder efficiënt aangezien je bij beide iets moet bijhouden (namelijk de kleur van een top of de verschillende sleutels in een top) en het verwijderen van een top complex kan worden. De AVL-boom vond ik al interessanter, aangezien de diepte bij deze boom O(log n) is. Hierdoor kunnen opzoekingen en verwijderingen efficiënt gebeuren. Om ervoor te zorgen dat de boom zichzelf tijdig herbalanceert, wordt er gekeken naar het verschil in dieptes van de kinderbomen.

Voor mijn eigen heuristiek heb ik mij vooral gebaseerd op de AVL-boom. Nadat een nieuwe top toegevoegd wordt, overloop ik de voorouders van de toegevoegde top en kijk ik na of de diepte kleiner of gelijk is aan het aantal toppen van die deelboom. Als er een top v is waarvoor dit niet geldt, dan gaat de boom zichzelf herschikken. Het herbalanceren gebeurt door top v te vervangen door de grootste sleutel uit zijn linkerdeelboom of door de kleinste sleutel uit zijn rechterdeelboom, afhankelijk van welke deelboom het meeste toppen bevat. Als de beide deelbomen evenveel toppen bevatten, wordt de deelboom gekozen waarbij de diepte groter is dan het logaritme van het aantal toppen.

Vanaf de top die top v vervangen heeft, wordt een nieuwe plaats gezocht voor top v. Daarna worden de voorouders van top v gecontroleerd op de diepte tot er opnieuw gebalanceerd moet worden (en dit proces herhaald wordt) of tot de wortel bereikt wordt.

Enkele nadelen aan deze heuristiek zijn dat de diepte vaak berekend wordt, dat toppen veel bezocht worden en dat er soms meerdere keren herbalanceerd moet worden bij 1 toevoeging. Het voordeel daarentegen is dat dit enkel gebeurt bij toevoegingen, waardoor verwijderingen en opzoekingen snel kunnen verlopen (met weinig bezochte toppen). Daarnaast heb ik het aantal bezochte toppen al een beetje beperkt door de diepte enkel te berekenen voor de voorouders.

Aangezien de 3 verschillende bomen allemaal vaak toppen moeten zoeken en een lijst van toppen moeten kunnen geven, kan dit leiden tot codeduplicatie.

Om dit te vermijden heb ik 2 aparte klassen toegevoegd. Met de klasse NodeSearcher kan je een gegeven top opzoeken in de boom met de methode "searchNode" en kan je het grootste linkerkind of het kleinste rechterkind bepalen met "findGreatestChild" en "findSmallestChild".

De klasse ListMaker vult een lijst recursief aan met alle toppen van de boom en wordt gebruikt om een iterator van de boom te geven.

**Benchmarks**

Bij de benchmarks heb ik voor SemiSplayTree en MyTree de tijd en het aantal bezochte toppen berekend bij toevoegingen, opzoekingen, verwijderingen en het slechtste geval. Voor mij is dit slechtste geval het toevoegen van sleutels 0 tot n in die volgorde.

De resultaten werden gegenereerd met zowel de Sampler klasse als de ZipfSampler klasse. Hoewel er verschillen aanwezig waren tussen de resultaten van beide klassen, waren deze niet zo groot en kan ik hierdoor één algemeen besluit maken.

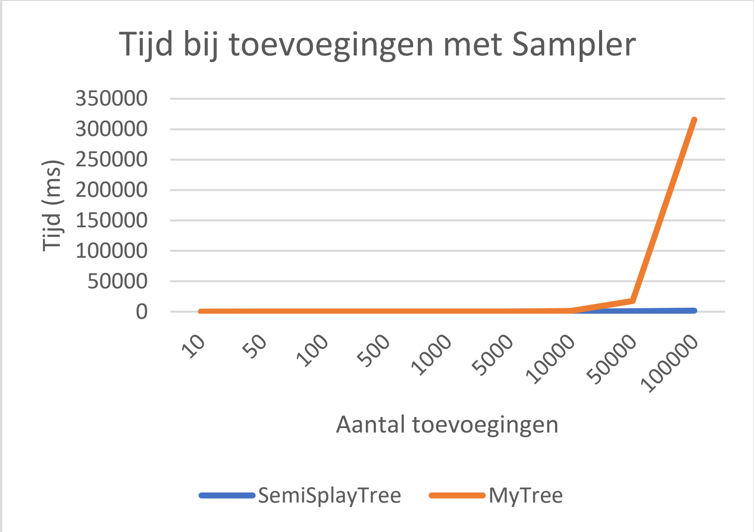
Als het gaat over het toevoegen van n sleutels is SemiSplayTree een betere keuze, aangezien deze bewerking veel sneller uitgevoerd wordt dan bij MyTree. Daarnaast is er een groot verschil in het aantal bezochte toppen. Dit komt doordat MyTree de boom nakijkt en eventueel herbalanceert bij het toevoegen van elke top.

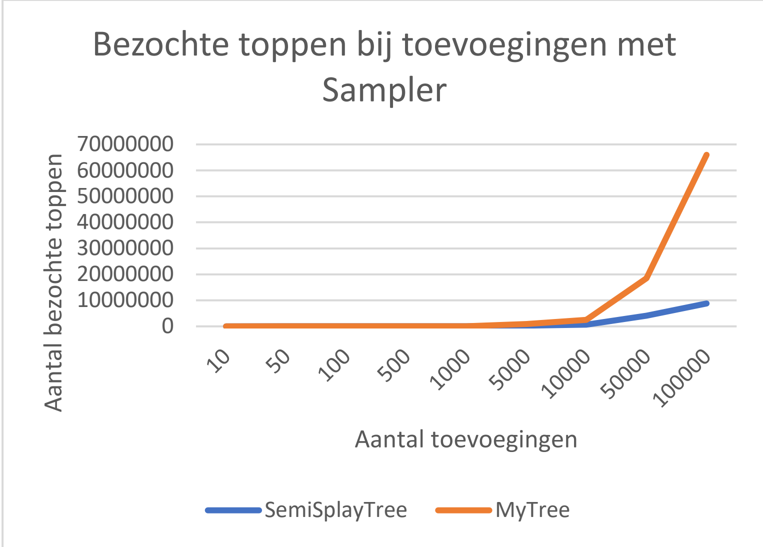
Daarentegen werkt MyTree sneller bij het opzoeken en verwijderen van toppen en worden er minder toppen bezocht. De oorzaak hiervan is dat MyTree na het herbalanceren een diepte heeft die hoogstens O(log n) is, terwijl dit bij de SemiSplayTree niet gegarandeerd kan worden.

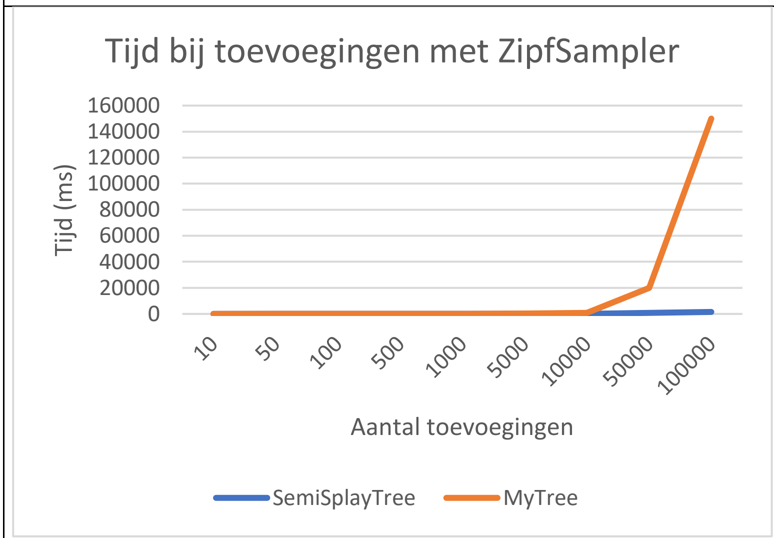
Bij het slechtste geval, sleutels 0 tot n toevoegen, zijn beide bomen redelijk snel en worden niet al te veel toppen bezocht.

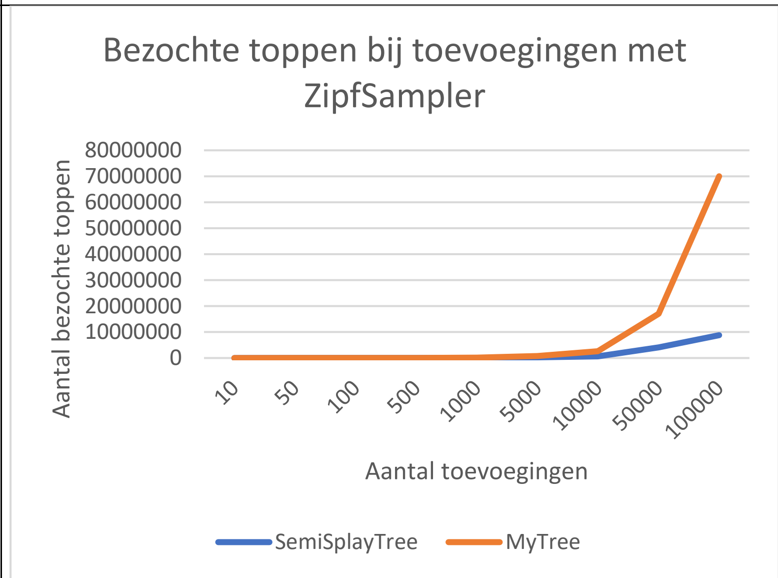
Hoewel MyTree efficiënter is bij meerdere bewerkingen, namelijk opzoek- en verwijderbewerkingen, is de inefficiëntie bij toevoegbewerkingen opvallender dan de inefficiënte van SemiSplayTree bij opzoekingen en verwijderingen. De uitvoeringstijd en aantal bezochte toppen van MyTree is bij toevoegingen immers veel groter dan die van SemiSplayTree bij opzoekingen en verwijderingen. Hierdoor is SemiSplayTree in het algemeen performanter en een betere implementatie van een binaire zoekboom.

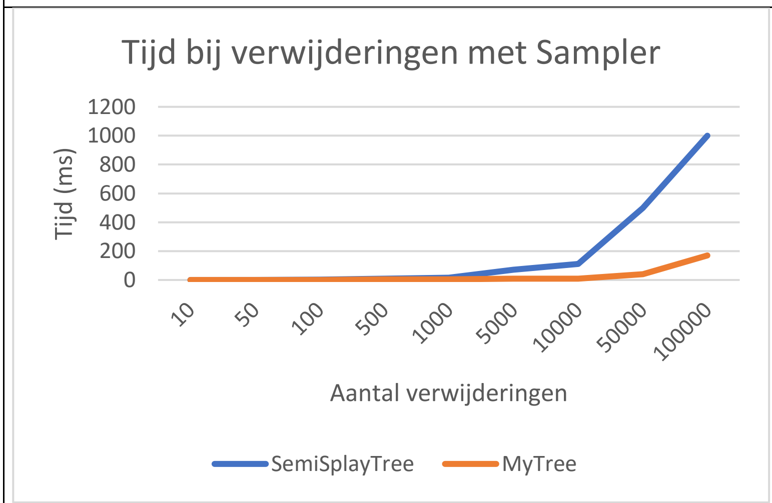
Janne Cools

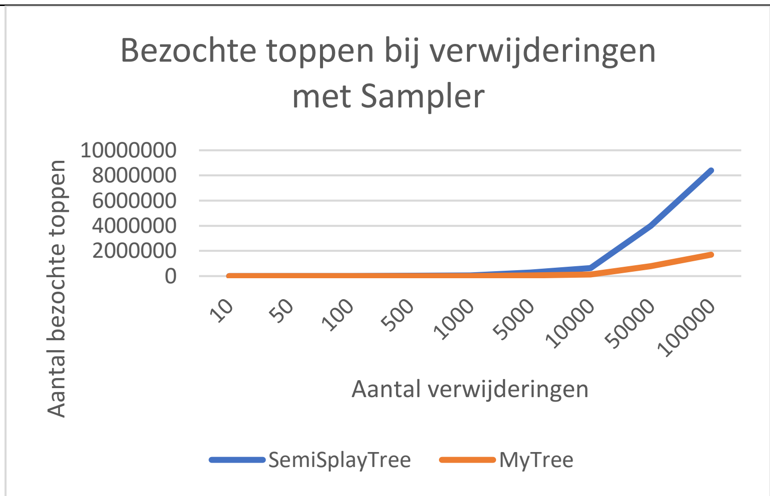

Tijd bij toevoegingen met Sampler
Tijd (ms)
350000
300000
250000
200000
150000
100000
50000
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal toevoegingen
SemiSplayTree MyTree


Bezochte toppen bij toevoegingen met Sampler
Aantal bezochte toppen
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal toevoegingen
SemiSplayTree MyTree

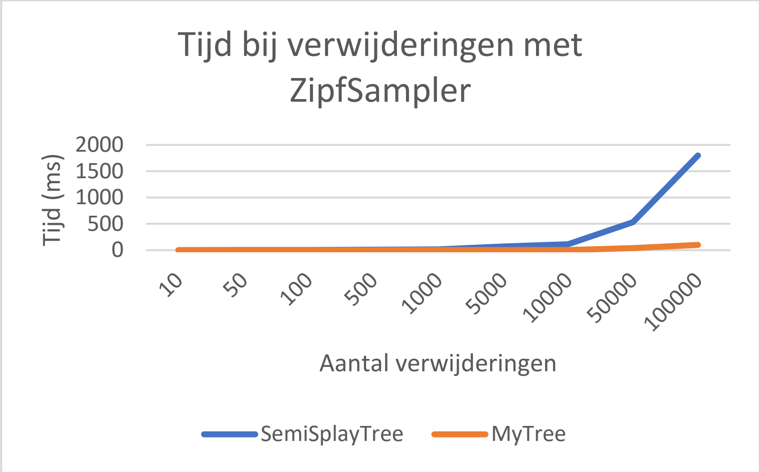

Tijd bij toevoegingen met ZipfSampler
Tijd (ms)
160000
140000
120000
100000
80000
60000
40000
20000
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal toevoegingen
SemiSplayTree MyTree

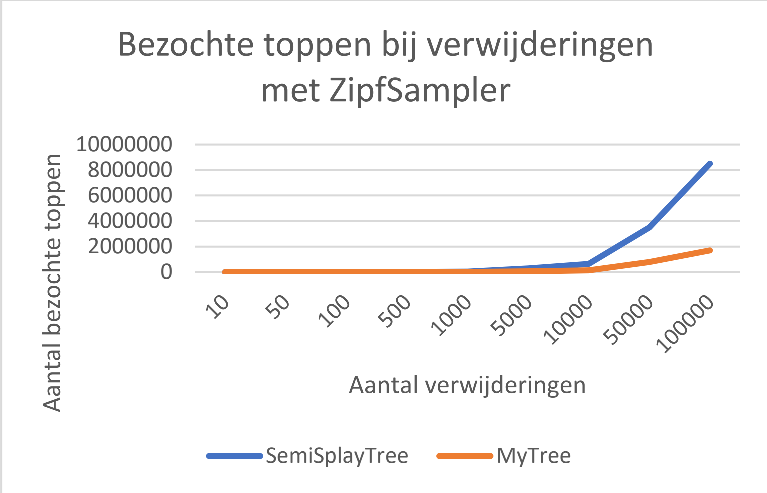

Bezochte toppen bij toevoegingen met ZipfSampler
Aantal bezochte toppen
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal toevoegingen
SemiSplayTree MyTree

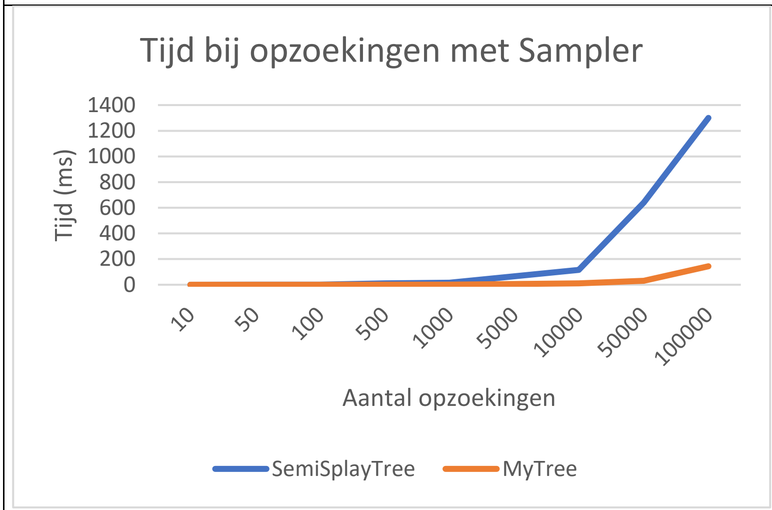

Tijd bij verwijderingen met Sampler
Tijd (ms)
1200
1000
800
600
400
200
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal verwijderingen
SemiSplayTree MyTree

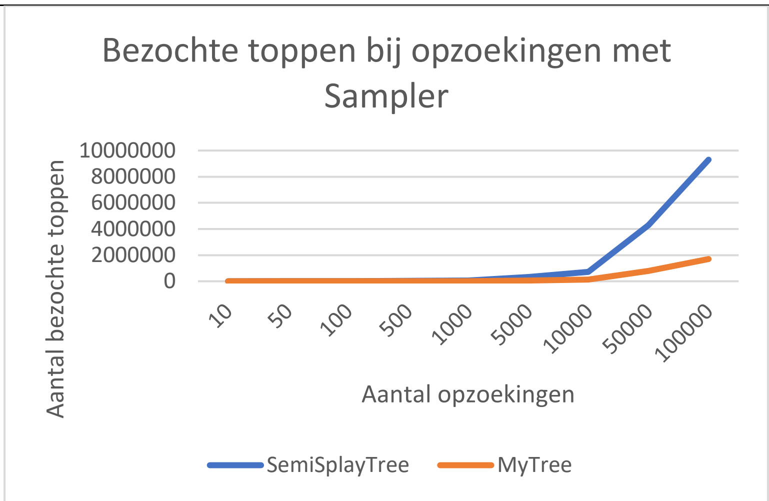

Bezochte toppen bij verwijderingen met Sampler
Aantal bezochte toppen
10000000
8000000
6000000
4000000
2000000
0
10 50 100 500 1000 5000 10000 50000 100000
Aantal verwijderingen
SemiSplayTree MyTree

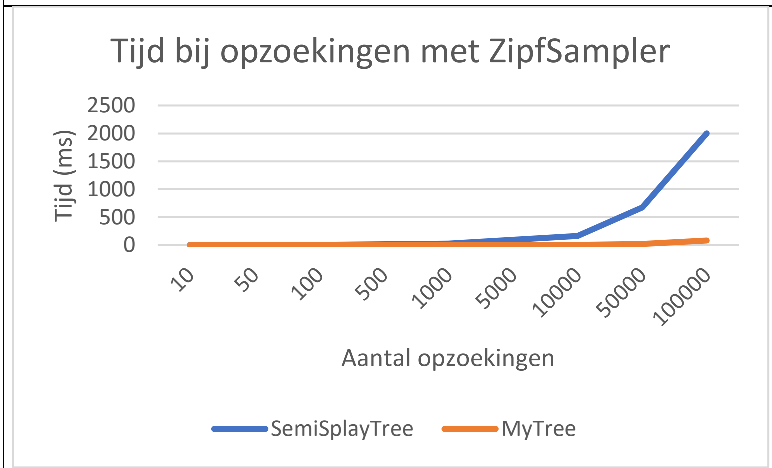
Janne Cools

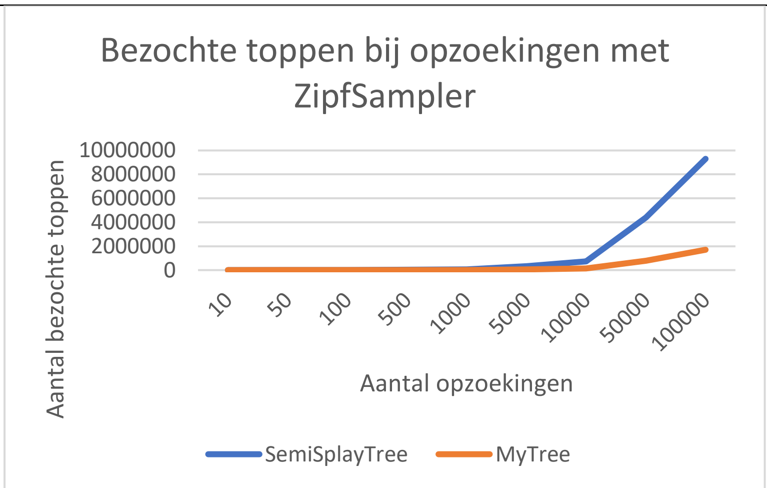

Tijd bij verwijderingen met ZipfSampler
Tijd (ms)
2000
1500
1000
500
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal verwijderingen
SemiSplayTree
MyTree


Bezochte toppen bij verwijderingen met ZipfSampler
Aantal bezochte toppen
10000000
8000000
6000000
4000000
2000000
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal verwijderingen
SemiSplayTree
MyTree

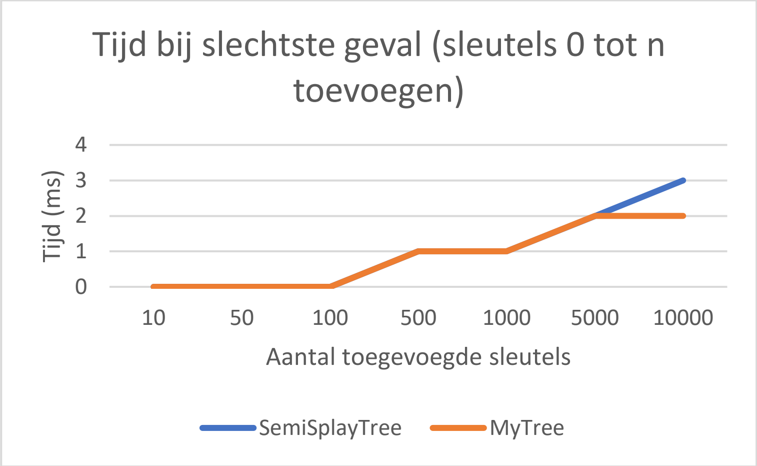

Tijd bij opzoekingen met Sampler
Tijd (ms)
1400
1200
1000
800
600
400
200
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal opzoekingen
SemiSplayTree
MyTree


Bezochte toppen bij opzoekingen met Sampler
Aantal bezochte toppen
10000000
8000000
6000000
4000000
2000000
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal opzoekingen
SemiSplayTree
MyTree

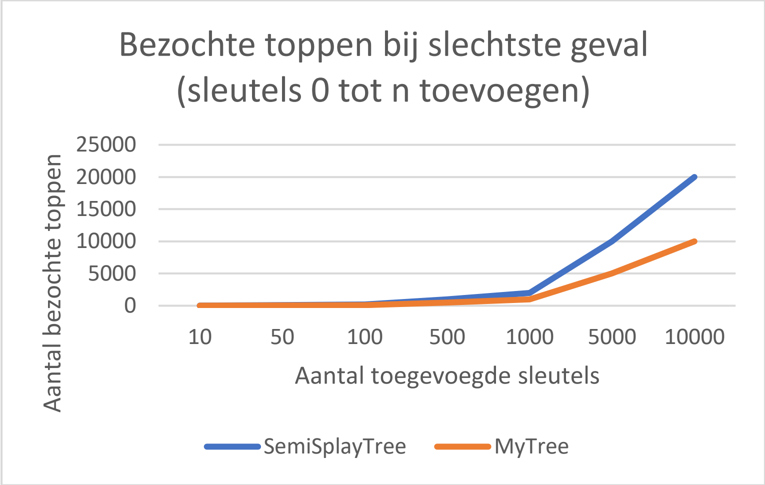

Tijd bij opzoekingen met ZipfSampler
Tijd (ms)
2500
2000
1500
1000
500
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal opzoekingen
SemiSplayTree
MyTree


Bezochte toppen bij opzoekingen met ZipfSampler
Aantal bezochte toppen
10000000
8000000
6000000
4000000
2000000
0
10
50
100
500
1000
5000
10000
50000
100000
Aantal opzoekingen
SemiSplayTree
MyTree

Janne Cools



Tijd bij slechtste geval (sleutels 0 tot n toevoegen)
Tijd (ms)
4
3
2
1
0
10
50
100
500
1000
5000
10000
Aantal toegevoegde sleutels
SemiSplayTree
MyTree

Bezochte toppen bij slechtste geval (sleutels 0 tot n toevoegen)
Aantal bezochte toppen
25000
20000
15000
10000
5000
0
10
50
100
500
1000
5000
10000
Aantal toegevoegde sleutels
SemiSplayTree
MyTree