Dit bestand hoef je niet te lezen, alle essentie ervan is in het verslag opgenomen

Inhoudsopgave

1 Algemeen

In deze deel bespreek ik op welke manier ik de trees benchmark. De benchmarkfuncties kun je terugvinden in `src/oplossing/benchmark/BenchmarkExecuter.java`. Voor het meten van de uitvoeringstijd gebruik ik het verschil in tijd (in milliseconden) tussen het moment na de uitvoering van een reeks bewerkingen en het moment ervoor. Voor het meten van het geheugen roep ik vóór het meten `System.gc()` aan, wat ervoor zorgt dat de JVM de garbage collector uitvoert. Vervolgens meet ik het totale heapgeheugen gebruik vóór en na de reeks bewerkingen met `rt.totalMemory() - rt.freeMemory()` en sla ik het verschil in gebruik op in kilobytes. Om de benchmarks iets nauwkeuriger te maken, voer ik elke test van een bepaalde grootte meestal meerdere keren uit en neem ik het gemiddelde van de uitvoeringstijd of het geheugengebruik. Daarbij wordt voor elke uitvoering dezelfde sampler gebruikt, maar een verschillende sample.

Wanneer we twee tree-implementaties vergelijken, worden voor de tests verschillende samplers gebruikt van dezelfde grootte. Dit is aanvaardbaar, aangezien een sampler willekeurige getallen genereert. Als we de test voldoende vaak uitvoeren met een voldoende grote elementgrootte, verkrijgen we voor beide bomen een eerlijke vergelijking. Het is immers vrijwel onmogelijk dat bijvoorbeeld bij tien uitvoeringen van een test met grootte 100.000, waarbij de elementen willekeurig worden gegenereerd, telkens exact die elementen worden gekozen die op de maximale diepte van de boom moeten worden toegevoegd.

De vergelijkingsomstandigheden zijn eerlijk, omdat de elementen die moeten worden toegevoegd altijd willekeurig worden gegenereerd met `Sampler`, en omdat voor beide trees evenveel toppen worden toegevoegd vanuit een identieke begintoestand (beide bomen leeg of niet leeg).

Ook te vermelden dat ik hier enkel de setting bespreek en resultaten toon. Conclusies op basis van de resultaten zijn getrokken in het verslag.

1.1 Gemiddelde benchmark

Vaak heb ik het over een gemiddelde benchmark, en om niet elke keer opnieuw te moeten uitleggen wat dat precies inhoudt, doe ik dat hier één keer. Deze benchmark is terug te vinden in `src/benchmark/AverageBenchmark.java` en verloopt op de volgende manier:

1. Een aantal toppen wordt toegevoegd.
2. Een aantal toppen wordt gezocht.
3. Een aantal toppen wordt verwijderd.
4. De bovenstaande stappen worden herhaald.

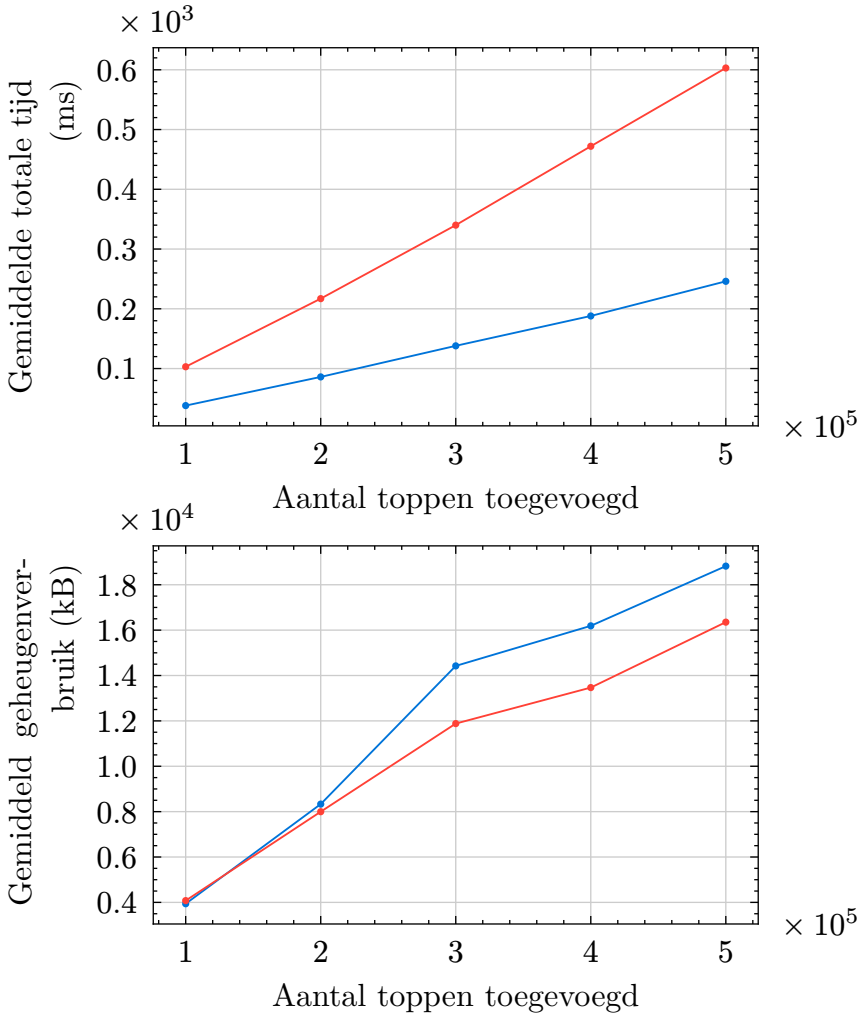
2 Semi-splay

2.1 Semi-splay met bijhouden van ouder in vergelijking met semi-splay met stack

Deze benchmark vergelijkt twee implementaties van semi-splay:

1. Elke bezochte top bij een bewerking wordt opgeslagen in een stapel, die vervolgens wordt gebruikt voor de semi-splay-operatie.
2. Bij elke nood de ouder wordt bijgehouden om op die manier het semi-splay-pad op te bouwen.

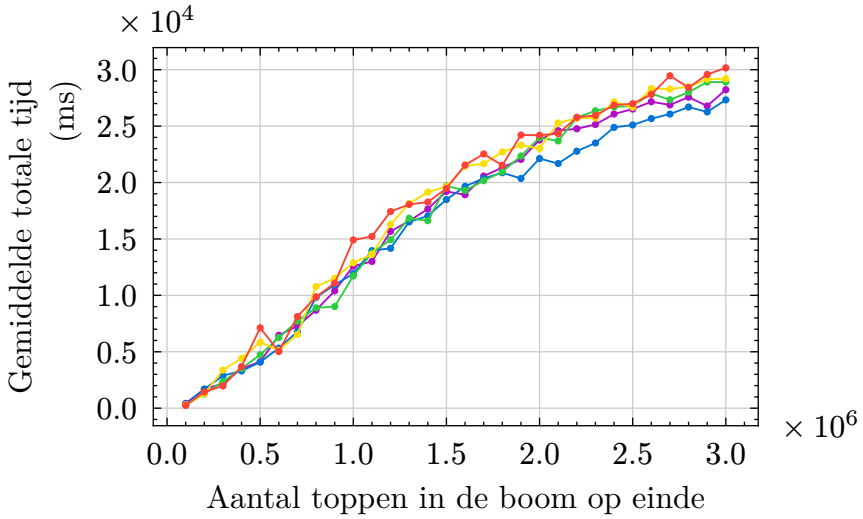
We doen een benchmark waarbij we een aantal elementen van groottes van 100.000 tot 500.000 met een stap van 100.000 toevoegen aan een initieel lege boom. Na elke toevoeging wordt de semi-splay-operatie uitgevoerd (indien de diepte groter is dan 1), waarna we het gemiddelde nemen van de uitvoeringstijden van de tien uitvoeringen. Zo is bijvoorbeeld de gemiddelde uitvoeringstijd ongeveer 200 ms bij het toevoegen van 200.000 toppen aan een initieel lege semi-splay-boom geïmplementeerd met een stack. Resultaten zijn te zien op Figuur 1.



Figuur 1: Semi-splay met bijhouden van ouder in blauw, semi-splay met stack in rood.

3 Treap

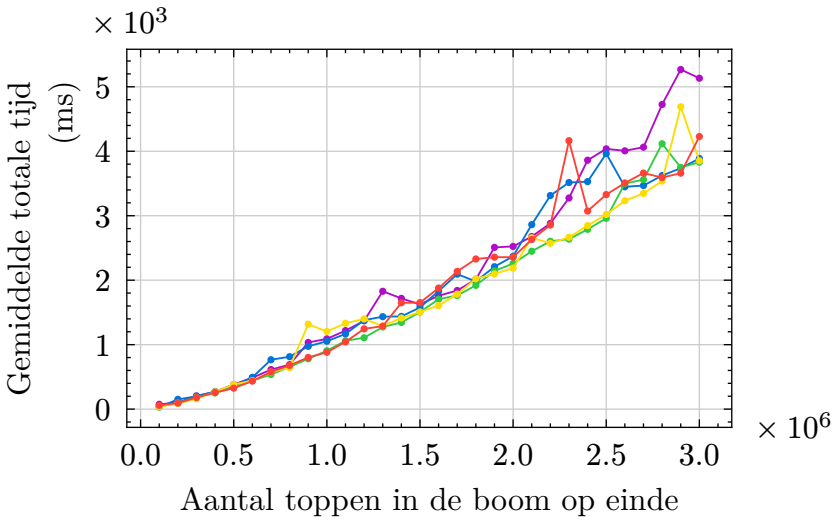
Hier benchmark ik grensen voor generatie van willekeurige prioriteit in `Treap` op uitvoeringssnelheid. Warbij grensen van 10.000, 100.000, 1.000.000, 10.000.000 en `Integer.MAX_VALUE`. Ik gebruik hiervoor gemiddlede benchmark. Resultaten zijn terug te vinden in Figuur 2.



Figuur 2: Bovengrens 10.000 in paars, 100.000 in blauw, 1.000.000 in groen, 10.000.000 in geel, `Integer.MAX_VALUE` in rood.

4 MyFrequencyTreap

Exact dezelfde benchmark als voor Treap voeren wij ook uit voor `MyFrequencyTreap`.



Figuur 3: Bovengrens 10.000 in paars, 100.000 in blauw, 1.000.000 in groen, 10.000.000 in geel, `Integer.MAX_VALUE` in rood.

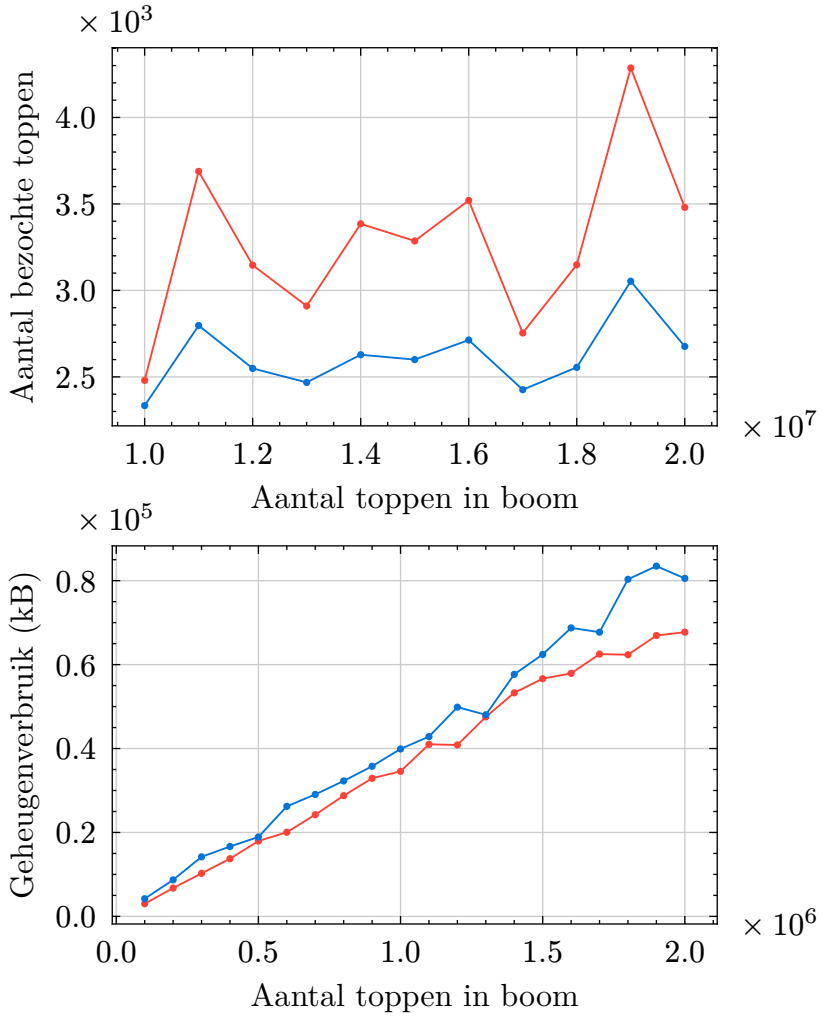
5 MyFrequencyTreap en LineairFrequencyTreap

In de volgende benchmark vergelijken wij `MyFrequencyTreap` en `LineairFrequencyTreap` in een scenario waar `LineairFrequencyTreap` slechter zou moeten presteren. De benchmark is uitgevoerd voor groottes van 10.000.000 tot 20.000.000 met een stap van 1.000.000, waarbij wij het aantal bezochte toppen meten.

De benchmark is terug te vinden in `src/benchmark/MyFrequencyTreapBenchmark.java` en verloopt op de volgende manier (waar n het totaal aantal elementen is):

1. Voeg n elementen toe
2. Start meting.
3. Kies een willekeurig blad.
4. Zoek hetzelfde element 2.000 keer.
5. Stop meting.

Het is belangrijk te vermelden dat, ongeacht welk blad willekeurig wordt gekozen, dit in beide bomen hetzelfde blad is. Bovendien hebben beide bomen vóór het starten van het zoeken dezelfde structuur. Dit is bereikt door dezelfde seed en bovengrens voor de generatie in beide bomen te gebruiken, en dezelfde seed voor beide benchmarks.



Figuur 4: LineairFrequencyTreap in rood, MyFrequencyTreap in blauw.

6 MyTreap, Treap, SemiSplayTree

Perfect scenario voor MyTreap

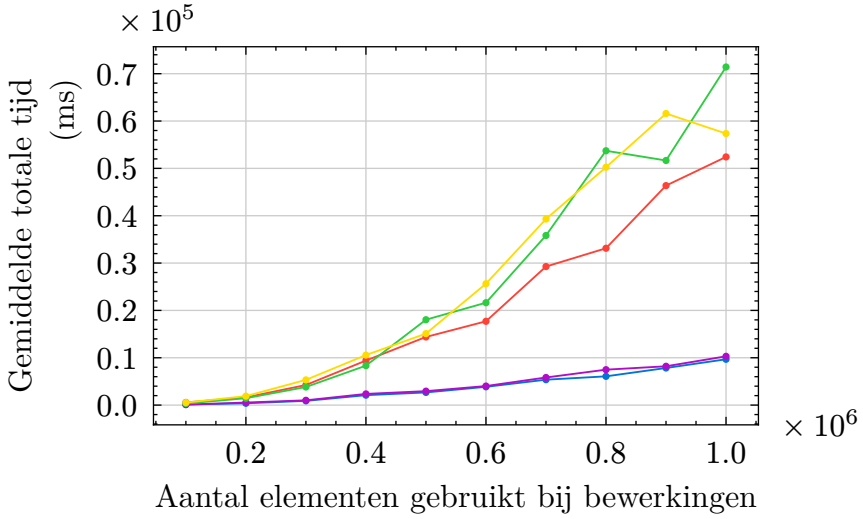
Hier benchmark ik `Treap`, `MyTreap` en `SemiSplayTree` in een perfect scenario voor `MyTreap` en `SemiSplayTree`. Benchmarks zijn uitgevoerd voor groottes van 100.000 tot 1.000.000. Voor elke grootte wordt de benchmark drie keer uitgevoerd op verschillende bomen (en lijsten van bladen), waarna het gemiddelde van de drie resultaten wordt genomen. De benchmark is terug te vinden in `src/benchmark/MyTreapBenchmark.java` en verloopt op de volgende manier:

1. Bouw een boom op van de gegeven grootte (zie hierboven voor de groottes).
2. Maak een lijst van toppen (meestal bladen) waarbij ook hun diepte wordt bijgehouden.
3. Start de tijdsmeting.
4. Zoek elke top uit de lijst `diepte + c` keer, waarbij `c` een fractie is van het totale aantal elementen.
5. Stop de tijdsmeting.

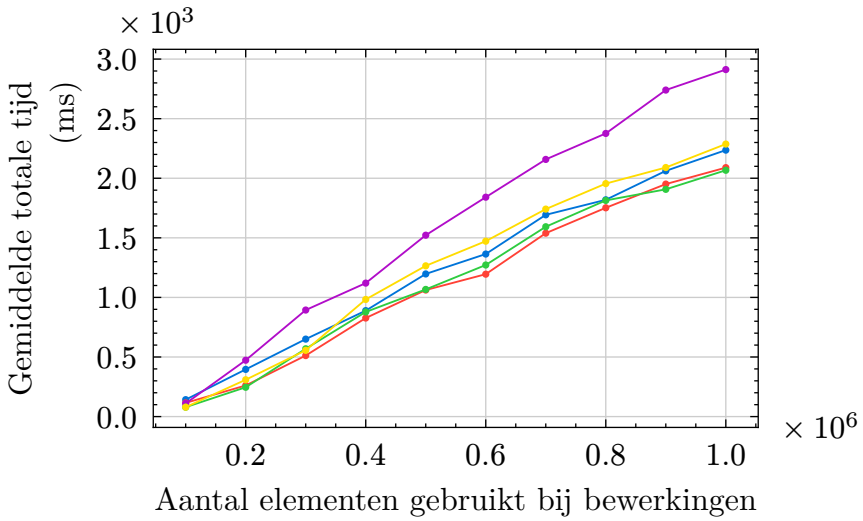
De resultaten van deze benchmark zijn te zien in Figuur 5.

Gemiddelde scenario

Hier benchmark ik `Treap`, `MyTreap` en `SemiSplayTree` in een gemiddeld scenario. De resultaten van deze benchmark zijn te zien in Figuur 6.



Figuur 5: Perfecte scenario voor MyTreap en SemiSplayTree. MyTreap in blauw, Treap in rood, SemiSplayTree in paars, LineairFrequencyTreap in groen en MyFrequencyTreap in geel.



Figuur 6: Gemiddelde scenario. MyTreap in blauw, Treap in rood, SemiSplayTree in paars, LineairFrequencyTreap in groen en MyFrequencyTreap in geel.