

# 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  $n-1$ ):

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

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

## Geïmplementeerde 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.

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 opzoeken 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 alle 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 aantal toppen.

Vanaf de top die top v vervangen heeft, wordt een nieuwe plaats gezocht voor top v. Daarna worden alle voorouders van top v gecontroleerd op de diepte tot er opnieuw gebalanceerd moet worden (en alles 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 meerdere keren herbalanceerd kan worden bij 1 toevoeging. Het voordeel daarentegen is dat dit enkel gebeurt bij toevoegingen, waardoor verwijderingen en opzoeken 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.

## Benchmarks

Telkens 10 keer bepaald aantal toppen toevoegen/verwijderen/opzoeken en het gemiddelde nemen van het resultaat

## OptimalTree

- Implementatie o.b.v. uitleg van filmpje  
[https://www.youtube.com/watch?v=vLS-zRCHo-Y&list=PLPKF\\_TcxpTJ40Ez42V8DIqm9IECUhb7MB&index=2&t=489s](https://www.youtube.com/watch?v=vLS-zRCHo-Y&list=PLPKF_TcxpTJ40Ez42V8DIqm9IECUhb7MB&index=2&t=489s)
- Gebruik 2-dimensionale array (rij i en kolom j met grootte array n+1)
- Diagonaal invullen met waarde 0 →  $\text{array}[i][j] = 0$  met  $i=j$
- Per lengte  $l = j - i$  alles invullen
  - Eerst  $l = 1 \rightarrow \{i,j\}: \{0,1\}, \{1,2\}, \{2,3\}, \dots, \{n-1,n\}$
  - Dan  $l = 2$  enzovoort
- Telkens elke top (in die lengte) een keer als wortel plaatsen en op die manier de kost berekenen → minimale kost is de kleinste berekende kost
- Boom herconstrueren o.b.v. tabel
  - Eerst  $\text{array}[0][n]$