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):

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

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

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.

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 bepaalde 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 start en stop, beide indexen) toe aan de boom als linker- of rechterkind van de ouder die meegegeven wordt als argument.

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 bom 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 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 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.

Daarom heb ik 2 aparte klassen toegevoegd om dit te vermijden. 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 berekenen met “findGreatestChild” en “findSmallestChild”.

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

Benchmarks

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