|  |
| --- |
| Project: Search Engine Design |
| Algorithms and data structures |
| Wannes Van Leemput en Armand Naessens |

|  |
| --- |
| Faculteit Ingenieurswetenschappen en Architectuur |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| [www.ugent.be](file:///D:\Documenten\12%20Dozijn\Universiteit%20Gent\Sjablonen\Stationery\www.ugent.be) |

|  |
| --- |
|  |

inhoudsopgave

[inhoudsopgave 2](#_Toc513900882)

[1 Inleiding 3](#_Toc513900883)

[2 Binary Tree 3](#_Toc513900884)

[3 Red Black Tree 4](#_Toc513900885)

[4 BK-Tree 4](#_Toc513900886)

[5 Ranking 4](#_Toc513900887)

# Inleiding

Voor het vak Algoritmen & Datastructuren hebben we dit semester een project uitgevoerd omtrent het ontwerpen van een zoekmachine. Dit doen we aan de hand van de in de les geziene leerstof in verband met binaire zoekbomen. Hieronder volgt een bespreking van de gebruikte algoritmes in dit project.

# Binary Tree

In een binaire boom heft elke knoop maximum 2 kinderen. De worst case tijdscomplexiteit voor een zoekopdracht van een binaire boom is O(n). In het normale geval zal de complexiteit O(log n) zijn. Dit omdat bij elk niveau in de boom het aantal knopen gemiddeld gezien verdubbeld. De hoogte van de boom in vergelijking met het aantal knopen is dus log n.

In het slechtste geval moet je elk element overlopen om de juiste knoop te vinden. Ook voor het toevoegen en verwijderen van een knoop is de worst case complexiteit gelijk aan O(n). Zie voorbeeld:

Search(2): de boom heeft 3 elementen en het kost 3 tijdscycli om 2 te vinden. De worst case complexiteit is dus O(n).

Insert(Childof2): opnieuw zullen we elke knoop moeten overlopen om een nieuw kind van 2 toe te voegen. De complexiteit is dus opnieuw O(n).

# Red Black Tree

Een red black tree heeft een tijdscomplexiteit van O(log n). Dit omdat de Red Black tree zelf balancerend is en er dus bijna altijd geldt dat elke knoop 2 kinderen heeft. De hoogte van de boom is dus log n ten opzichte van het aantal knopen. De geheugen complexiteit van de Red Black Tree is O(n).

Een goed alternatief voor de Red Black Tree zou een B Tree kunnen zijn. Een B tree is ook zelf balancerend en het verschil tussen een Binary search tree en een B tree is dat een B tree meer dan 2 kinderen mag hebben. In elke knoop zitten een bepaald aantal “keys”. Deze keys doen dienst als splitsingswaarden van waaruit de verschillende subbomen ontstaan.

Een ander alternatief zou de AVL-tree kunnen zijn. Deze zijn erg gelijkaardig aan de Red-Black trees met het verschil dat voor look-up opdrachten de AVL-tree in het algemeen wat sneller is daar deze strikter gebalanceerd zijn dan Red Black Trees.

# BK-Tree

De complexiteit hangt van meer af dan alleen het aantal nodes, in dit geval het aantal woorden in het woordenboek. Als we stellen dat de tolerantielimiet gelijk is aan 2 dan kunnen we volgende complexiteit vaststellen: O(L1\*L2\*log n) met n het aantal woorden in het woordenboek, L1 de gemiddelde lengte van woorden in het woordenboek en L2 de lengte van het aantal schrijffouten. Algemeen zijn L1 en L2 klein.

Er is O(m\*n) opslagruimte nodig voor de dynamic programming version van de Levenshtein afstand??

<how to improve?>

<what is the time comparing query ..>

# Ranking

Voor het opslaan van tussentijdse resultaten gebruiken wij de map data structuur. Een map bestaat uit een unieke verzameling keys die elk een bepaalde waarde met zich mee dragen. In dit geval is een key een oplossing ( een woord ) en de value is dan de score van dat woord.