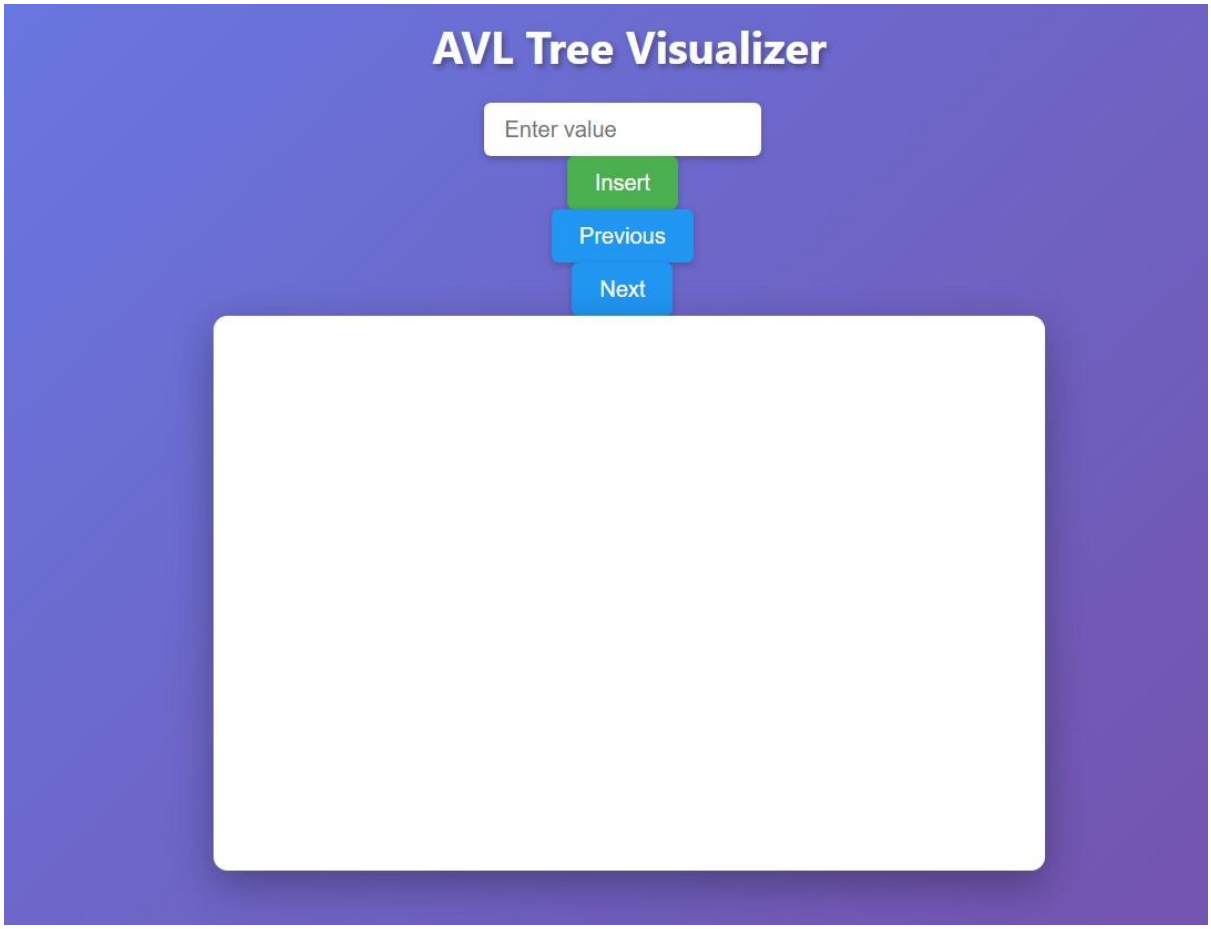
Idris Azmi Ata Isci
Erhvervsakademi København, Datamatiker
Datastrukturer & algoritmer
8/12-25

AVL-Tree

Idris Isci – cph-ii38@stud.ek.dk - @mridrisisci
Link til Github-projekt: https://github.com/mridrisisci/DSA-AVL-Tree
Link til deployet app: https://mridrisisci.github.io/DSA-AVL-Tree/

Screenshot af den kørende applikation (lokal server)



Generel beskrivelse af projektet

Projektet handler om at implementere en algoritme og visuelt fremvise hvordan algoritmen sker trin for trin for en given operation. Min visualisering lægger vægt på at fremvise processen for indsættelsesoperationen for et AVL-tree. Denne proces ser man trin for trin med udgangspunkt i hvordan logikken er skrevet i AVLTree.js.

Man kan anvende webapplikationen ved at køre sin webserver lokalt på sin maskine. Herefter besøger man webstedet – typisk localhost:5500. Man bliver mødt af et blankt kanvas og 3 knapper – insert, previous og next.

Disse knapper bruges til at udforske funktionaliteten af indsættelsesoperationen. Man kan tage udgangspunkt i eksemplet 30,20,10. Man indsætter dem enkeltvis i den rækkefølge, det er skrevet. Resultat af indsættelsen for eksemplet ovenfor skaber en BST-struktur. Ved at klikke på next og previous knapperne kan man følge med trin for trin i hvordan BST-strukturen konceptuelt balancerer sig. Formålet med balanceringen er at opnå et selvbalancerende binært søgetræ aka. AVL-tree.

Uddybende beskrivelse af projektet

Projektet tager udgangspunkt i et AVL-tree, som er et selvbalancerende binært søgetræ. Et almindeligt binært søgetræ bruger en worst case *o(n)* da datastrukturen kan degenerere til en datastruktur der ligner linked list og derved ende med en tidskompleksitet af Big-O O(n).

Løsningen til dette er et AVL-tree, der garanterer en tidskompleksitet af Big-O O(log n). Dette opnås ved at et AVL-tree sikrer at alle noder er i balance, hvilket medfører højere effektivitet, hurtigere søge tid og en logaritmisk højde for hele træet.

Ved indsættelse sammenligner algoritmen den nye node med roden for at identificere hvilken vej gennem træet algoritmen skal traversere. Traverseringen fortsætter indtil der findes en tom plads. Herefter traverserer algoritmen fra bladet op til rodet igennem samme sti og beregner balancefaktoren for hver node rekursivt. Dette sikrer invarianten ikke er brudt. Hvis et brud er fundet, vil en rotation være nødvendig.

Overvejelser om tid- og pladskompleksitet

BST vs AVL-tree

Et selvbalancerende binært søgetræ har en worst case O(log n) da datastrukturen adskiller sig fra et almindeligt binært søgetræ ved at udføre rotationer efter hver sletning og indsættelse. Dette sikrer at træets højde er logaritmisk. Når man søger, sletter eller indsætter i træet, er worst case tidskompleksitet O(log n).

AVL-træet sikrer en hurtig performance ved udførelse af rotationer. Uanset størrelsen på træet er tidskompleksiteten for at opdatere referencer mellem noder konstant tid O(1) da antallet af disse ombytninger er et konstant antal.

Pladskompleksitet

Pladskompleksitet for en node er konstant tid Big-O O(1) da en node gemmer et fast antal af værdier heraf højden, referencer til næste og forrige node samt sin egen værdi. Men samlet set vil pladskompleksiteten være O(n) da hukommelsesforbruget afhænger af inputstørrelsen og vokser derfor lineært med inputtet.

Rekursion og stack-plads

AVL-træet garanterer logaritmisk højde og grundet at AVL-træet implementeres rekursivt garanterer det også at man har mere plads i sin stack. Stackken er afhængig af højden for træet og herom kan man garantere tidskompleksiteten for stackken.

Big-O

Datastrukturen og algoritmen er et effektivt valg og forudsigeligt grundet garantien for at worst case af tidskompleksitet er O(log n). ***Selvom at man har mindre pladskompleksitet så garanterer et AVL-tree en logaritmisk tidskompleksitet og vedligeholder denne.***

Implementering af algoritmen i koden

Balancering - invariant

Når invarianten er brudt, skal invarianten genoprettes igen. Det acceptable interval for invarianten er [-1,0,1] og en balancefaktor uden for dette er uacceptabelt, da det bryder AVL-træets garanti om en worst case af Big-O O(log n). Først identificerer algoritmen hvilken rotationscase, der er opstået, heraf en af følgende LR,RL,LL,RR. Så, udføres der rotation på den ubalancerede node, referencer ombyttes på de berørte noder. Når strukturen er opdateret fortsætter algoritmen rekursivt op gennem træet og sikre at invarianten ikke er brudt i resten træet.

Rotationscases

Hvis en balancefaktor for en node overstiger 1 er noden venstretung og ender med en rotationscase der enten er LL/LR (Left-Left case eller Left-Right case). Hvis en nodes balancefaktor er mindre end -1 er noden højretung og kan enten have en RR/RL-case. (Right-Right case eller Right-Left case).

I konteksten af en rotationscase har denne operation en tidskompleksitet af Big-O O(1) konstant tid. Dette skyldes at selve operationen udelukkende kræver at ombytte referencerne for de relevante noder. Denne ombytning af referencer genopretter balancen højere oppe i træet.

Visualisering og MVC (Model-View-Controller)

Logikken for algoritmen og lageret der indeholder visualiseringen er adskilt i den forstand at AVLTree.js håndterer datastrukturen og algoritmen mens visualiseringen håndterer at man kan følge med trin for trin og se konceptuelt hvordan indsættelse af noder og rotationer sker i et selvbalancerende binært søgetræ.

Implementering af LR/RL-cases

Den konceptuelle visualisering af LR/RL-cases er ikke implementeret i koden. Dette var et designvalg baseret på at rotationerne for disse rotationscases kan anses for at være en forlængelse af LL/RR-cases.

De grundlæggende rotationscases omhandler en Left-Left-case og en Right-Right-case. En LR/RL-rotation er operationer der sker efterfølgende af førnævnte rotationscases. Logikken for sådanne rotationer er inkluderet i projektet.

Pseudokode for algoritmen

Pseudokoden skaber et mere overordnet overblik af hvordan algoritmen skal håndtere logikken for at beregne balancefaktor for hver node, at der skal være rekursion ved indsættelse af en node for at sikre at træet er i balance.

De relevante funktioner der er udarbejdet ud fra pseudokoden ses som følger:

1. balance()
2. insertNode()
3. rightRotate()
4. leftRotate()
5. rotateLeftRight()
6. rotateRightLeft()

- Pseudokoden for indsættelsesalgoritmen

```
INSERT(node, value):
 hvis node er null: return ny node

hvis value < node.value:
   node.left = INSERT(node.left, value)
 ellers:
   node.right = INSERT(node.right, value)

 opdater height(node)
 balance = height(left) - height(right)

 hvis balance er ±2:
   udfør passende rotation

 return node
```

- Pseudokoden for rotationerne

```
 node.height = 1 + max(HEIGHT(node.left), HEIGHT(node.right))

 balance = HEIGHT(node.left) - HEIGHT(node.right)

 hvis balance > 1 og value < node.left.value: // LR-case
   return RIGHT_ROTATE(node)

 hvis balance < -1 og value > node.right.value: // RL-case
   return LEFT_ROTATE(node)

 hvis balance > 1 og value > node.left.value: LL-case
   return RIGHT_ROTATE(node)

 hvis balance < -1 og value < node.right.value: RR-case
   node.right = RIGHT_ROTATE(node.right)

 return node
```

Link til inspiration

Wikipedia side om algoritmen
https://en.wikipedia.org/wiki/AVL_tree

GeeksForGeeks om indsættelse
https://www.geeksforgeeks.org/dsa/insertion-in-an-avl-tree/

Big-O notation – forklaring af O(log n)
https://www.youtube.com/watch?v=wjDY5RbILno

AVL-tree – Indsættelse og rotationer
https://www.youtube.com/watch?v=jDM6_TnYIqE&t=1268s

Eksempel på implementering af AVL-tree algoritme
https://www.tutorialspoint.com/AVL-Tree-class-in-Javascript

Gennemgang af operationerne i et AVL-træ
https://www.youtube.com/watch?v=XeYQ2jSa2cI