

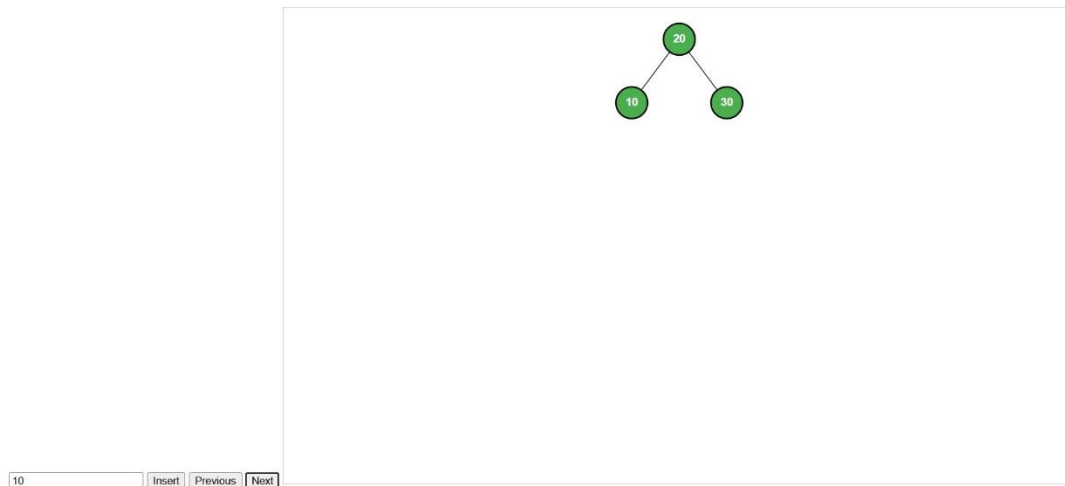
# AVL-Tree

Idris Isci – [cph-ii38@stud.ek.dk](mailto:cph-ii38@stud.ek.dk) - @mridrisisci

Link til Github-projekt: <https://github.com/mridrisisci/datastruktur-algo-eksamen>

## Screenshot af den kørende applikation (lokal server)

### AVL Tree Visualizer



## Generel beskrivelse af projektet

Dette projekt handler om at demonstrere visuelt hvordan algoritmen der er udviklet af de 2 russere Georgy Adelson-Velsky and Evgenii Landis i 1962. Demonstrationen lægger vægt på at vise hvad der sker trin for trin. Projektet lægger vægt på indsættelse og balancering, men AVL-tree understøtter også operationerne sletning og søgning

Der er udarbejdet en visuel demonstration af en indsættelsesalgoritme og sikret at man kan se hvad der sker trin for trin. Man kan navigere i webapplikationen ved at bruge knapperne "previous" og "next" og "insert". Det anbefales at man bruger det klassiske eksempel med 30,20,10. Dette gør man ved at indsætte de vilkårlige værdier ved at bruge "insert"-knappen. Hertil vil man se disse værdier blive oprettet som noder i diagrammet.

Man vil se at der sker en rotation automatisk i det man indsætter alle værdierne. Efter indsættelse af værdierne bruger man knapperne "previous" og next til at se før og efter tilstand af indsættelsesalgoritmen.

Diagrammet efter rotationen vil have rykket værdien 30 ned til højre, så værdien ligger på rodens højre barn og værdien 20 vil nu ligge øverst og til sidst vil værdien 10 ligge på rodens venstre barn, da tallet 10 er lavere end 20.

## Uddybende beskrivelse af projektet

### Uddybende forklaring af indsættelsesalgoritmen

I visualiseringen vil man se at ved at indsætte 3 vilkårlige værdier der vil kræve en rotation f.eks. 30,20,10. Herom vil værdien 30 ligge øverst, når man indsætter værdien 20, vil den ligge på rodens venstre barn da tallet 20 er lavere end 30. Når værdien 10 indsættes vil rotationen ske omgående og automatisk. Dette sker grundet en LL (Left-Left) case. Da 10 vil lande på venstre barn af venstre barn vil træet være venstre tungt. Dette gør at man skal lave en right-rotation for at opnå balance igen.

Efter roteringen vil man også skulle beregne højden af noden og træet igen.

### Det man ser i applikationen er:

1. Det selvbalancerende BST roterer ved indsættelse
2. Højderne bliver opdateret
3. Balancefaktoren afgør hvornår træet bliver opdateret
4. Før- og efter tilstand for hver rotation (vha. next/previous-knapperne)

Et AVL-tree er en variant af Binary Search Tree og har den forskel at AVL-tree bruger en balancefaktor til alle noder og har en automatisk rotation for at undgå ubalance i træet. Denne algoritmetype er særligt brugbar til:

1. databaser, da det kræver hyppig søgning,
2. Indeksering,
3. Applikationer med højt forbrug af hukommelse,
4. Spilapplikationer,
5. Andre applikationer der kræver opdatering i realtid.

Algoritmetypen gør brug af  $O(\log n)$  notationen, hvor 'O' står for kompleksiteten og 'n' står for antal af noder og 'log' forkortelsen for logaritme som fortæller at denne algoritmetype bruger altså den logaritmiske tid for  $O(\log n)$ .

Dette betyder at højden af AVL-træet er proportionelt med logaritmen til antallet af noder.

**Lidt om rotationer. Der findes 4 slags rotationer heraf:**

1. RR (Right rotation)
2. LL (Left rotation)
3. LR (Left-right rotation)
4. RL (Right-left rotation)

## Overvejelser om tid- og pladskompleksitet

Den store fordel ved denne algoritme er dens evne til at selvbalancere og undgå ubalance. I modsætning til BST. Dette effektiviserer søgning og indsættelse grundet logaritmisk tid. Ulempen hertil er at det er mere komplekst at opsætte, da algoritmen hele tiden skal opdatere højder og udføre rotationer ved ubalance.

I best case bruger man notationstypen  $O(1)$  ved søgning når det angår tidskompleksitet. Sådan et scenarie vil ske hvis værdien man søger efter er lig med roden af træet. Dog ved andre tilfælde vil man bruge notationstypen  $O(\log n)$  da træet vil have behov for balancering under søgningen.

Fx  $50 = \text{root.value} > \text{true}$

## Implementering af algoritmen i koden

### Indsættelse

Indsættelsesoperationen begynder ved *insert(value)* funktionen. Denne funktion modtager værdien fra front-enden. Derefter modtager *insertNode()* funktionen denne værdi. Funktionen bruger næsten samme logik som et BST. Forskellen er at den gør brug af en automatisk beregning af balancefaktoren ved en ny indsættelse af en node.

**Processen i *insertNode()* funktionen er:**

- Funktionen checker om noden i forvejen findes, hvis ikke bliver noden oprettet
- Funktionen checker om noden skal oprettes som højre barn eller venstre barn og returnerer noden
- Funktionen beregner den nye højde for noden
- Funktionen henter balancefaktoren vha. *getBalance()*
- Funktionen beregner ny balancefaktor.
- Funktionen returnerer til sidst noden

**Processen i *rightRotate()* funktionen er:**

- Funktionen modtager en root node
- Funktionen laver en ny root node og sætter den lig med venstre barn
- Funktionen sletter derefter venstre barn
- Funktionen roterer derefter den gamle root over til højre barn
- Funktionen sætter til sidst venstre barn af venstre barn til blot at være barn af den nye root node
- Funktionen returnerer den nye root node

Denne proces er tilsvarende for funktionen *leftRotate()* funktionen

**Processen for *getBalance()* er:**

- Funktionen modtager en node som parameter
- Funktionen beregner højden for træets venstre barn og trækker summen af træets højde for højre barn og returnerer denne sum.
- Denne sum bruges til at afgøre om der skal ske en rotation
- Hvis noden ikke findes returneres værdien 0

**Processen for *getHeight()* er:**

- Funktionen modtager en node som parameter
- Funktionen returnerer nodens højde, hvis noden findes.
- Hvis noden ikke findes, returnerer funktionen værdien 0

**Implementering af rotationerne**

- **Ved LL-case (venstre-venstre)**
  - Betingelse:  $balance > 1$  og  $value < node.left.value$
  - Løsning: `rotateRight(node)`
- **Ved RR-case (højre-højre)**
  - Betingelse:  $balance < -1$  og  $value > node.right.value$
  - Løsning: `rotateLeft(node)`
- **Ved LR-case (venstre-højre)**
  - Betingelse:  $balance > 1$  og  $value > node.left.value$
  - Løsning: `rotateLeft(node.left) -> rotateRight(node)`
- **Ved RL-case (højre-venstre)**
  - Betingelse:  $balance < -1$  og  $value < node.right.value$
  - Løsning: `rotateRight(node.right) -> rotateLeft(node)`

**Søgning**

Ved søgning fungerer det ligesom i et BST. Her vil algoritmen check om værdien man søger efter, er højre end eller lavere end den pågældende node. Hvis værdien man søger efter fx 20 er lavere end den pågældende værdi på (nodens plads) fx 30 så vil algoritmen søge i venstre barn og omvendt hvis værdien er højere end 30 så vil man søge i højre barn.

## Sletning

Denne operation følger også samme princip som et BST. Hvis den ønskede slettet værdi findes på det pågældende blad, bliver bladet slettet. Hvis bladet har et barn, vil barnet overtage forælderens plads. Hvis bladet har 2 børn, vil dette barn også rykke sig med op.

## Pseudokode for algoritmen

Ved indsættelsesoperationen vil jeg begynde i rodet af træet, check om pladsen er tom, hvis den er tom, vil jeg oprette noden på den plads. Hvis pladsen ikke er tom, vil jeg check om værdien er mindre end værdien på den pågældende plads. Hvis den er, vil jeg placere værdien i en ny node til venstre og hvis værdien er højere, vil jeg placere den til højre. Hvis værdien er lig med pladsens værdi, skal værdien ikke indsættes igen som en dublet.

Jeg vil beregne balancefaktoren ved at trække summen af højden af venstre side fra summen af højden af højre side. Dette vil jeg gøre med mål om at sikre at værdierne ikke overstiger -1, eller 1 og hvis de gør, skal jeg genbalancere træet ved at bruge rotationer.

Jeg vil sikre at man kan lave alle 4 rotationer (LL,RR,LR,RL).

## Link til inspiration

### Wikipedia side om algoritmen

[https://en.wikipedia.org/wiki/AVL\\_tree](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](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>