AVL-träd

Definition Ett AVL-träd är ett binärt sökträd där det för varje nod gäller att skillnaden i höjd mellan nodens vänster och höger subträd är högst 1.

AVL-träd kallas också för *höjdbalanserade* träd.

Vi skall här visa att höjden i ett AVL-träd med n noder är $\Theta(\log n)$ vilket garanterar att en sökning aldrig tar mer tid än så. Vi skall också beskriva en inläggningsalgoritm som underhåller AVL-egenskapen och som också tar $\Theta \log n$ tid.

Inlägg i AVL-träd

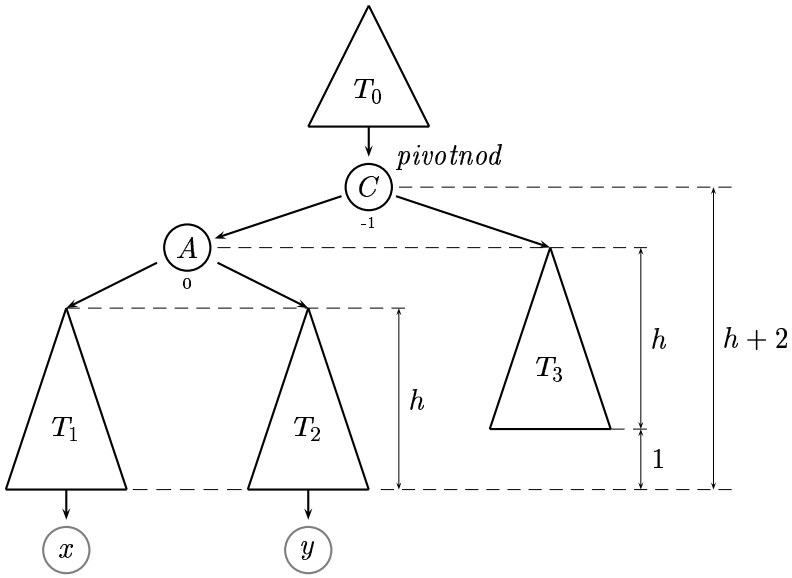
Varje nod förses med en *balansindikator* som anger skillnaden i höjd mellan höger och vänster subträd. Tillåtna värde är således -1 (vänster subträd högre än höger subträd), 0 (subträden är lika höga) och +1 (höger subträd är högre än vänster subträd).

Ett inlägg i ett AVL-träd börjar som ett vanligt BST-inlägg dvs man söker sig längs en väg från roten ner till den plats som det nya elementet skall ha. Den nya noden blir som vanligt ett löv i trädet.

Därefter följer man samma väg tillbaka upp mot roten. Så länge man passerar noder med balansvärdet 0 skall värdet ändras till 1 eller -1. Den första nod man påträffar som har balansfält skilt från noll kallar vi *pivot*-nod.

Om inlägget gjordes i pivotnodens lägre subträd har balansen förbättrats. Balansfältet sättes till noll och det hela är klart — ingenting behöver göras ovanför denna punkt.

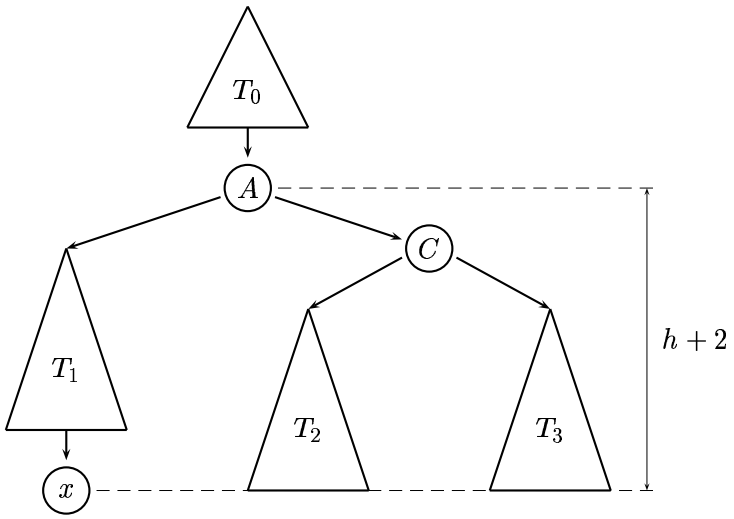
Om inlägget gjordes i det högre av subträden så måste AVL-egenskapen återställas genom en omstrukturering. Antag att inlägget gjordes i det vänstra subträdet. Fallet med inlägg i höger subträd hanteras symmetriskt) Den nya noden kan då ha hamnat antingen i pivotnodens vänsterbarns vänstra subträd (kallat T_1 i figuren nedan) eller i dess högra subträd (kallat T_2 i figuren nedan). Situationen beskrivs då av följande figur där x är nya noden i fall 1 och y är den nya i fall 2.:



(Balansvärdena i figuren är de värden som gällde *innan* inlägget gjordes.)

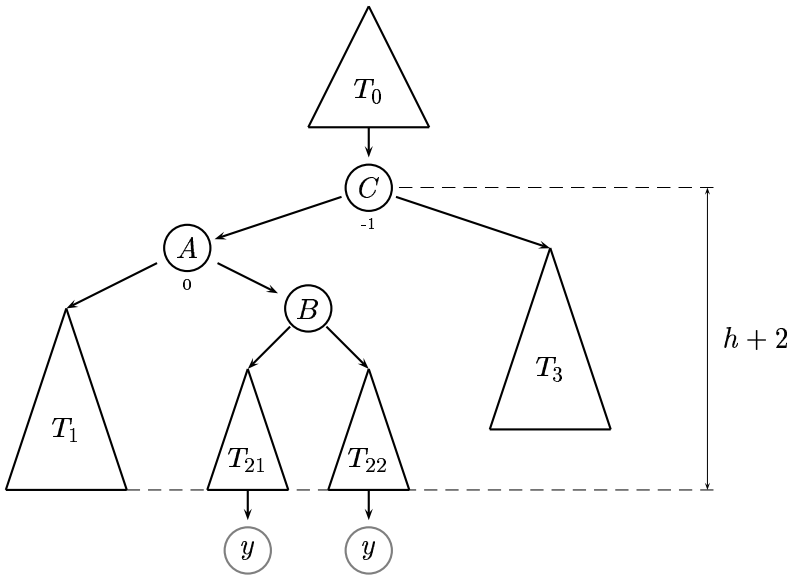
Vi tittar först på hur vi skall hantera fallet att inlägget gjordes i T_1. Problemet löse då genom att lyfta upp noden A till pivotnodens plats medan pivotnoden (C) åker ner ett steg åt höger. På det sättet återställs det brutna balanskriteriet vid pivotnoden. Dessutom bevaras den inbördes ordningen mellan noderna vilket är nödvändigt för behålla BST-egenskapen.

Denna operation kallas för en *enkel högerrotation* och resulterar i följande situation:

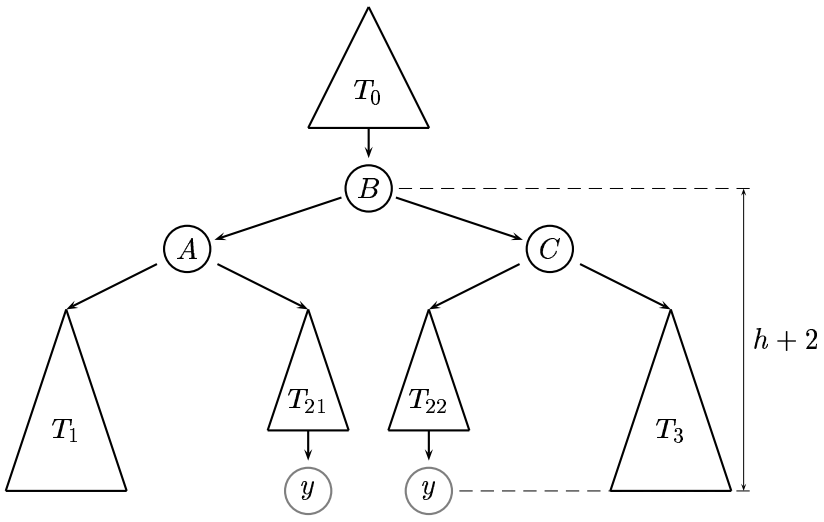


Observera att höjden av trädet under pivotnoden (noden som nu innehåller A är $h+2$ dvs detta delträd har samma höjd som det hade före inlägget. Detta innebär att ovanför denna punkt behövs ingen omstrukturering göras och inga balansfält modifieras.

Vi skall nu se på det andra fallet när den nya noden kom i T_2. För att hantera detta måste vi öka upplösningen i figuren så att vi även ser roten till T_2 som vi kallar B:



Det spelar ingen roll i vilket av de två subträden T_{21} eller T_{22} den nya noden y hamnar — båda fallen hanteras med en *dubbel högerrotation* som lyfter upp B till pivotnodens plats:

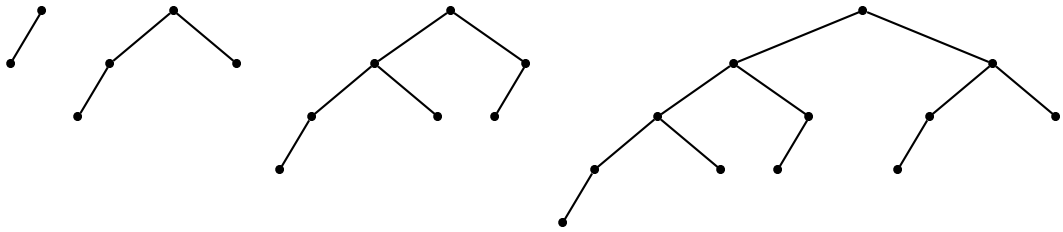


Även här är höjden av trädet med pivotnoden som rot oförändrad efter inlägget så att ingenting behöver göras högre upp i trädet.

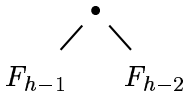
Hur högt kan ett AVL-träd bli?

För att se vad den den största höjden i ett AVL-träd med n noder kan bli betraktar vi en speciell följd av maximalt dåliga AVL-träd.

Låt F_h var ett AVL-träd med h nivåer konstruerat med *så få noder som möjligt*. Således är F_0 det tomma trädet och F_1 trädet med en nod. Därefter kommer träd med följande utseende:



Träden är inte unika; vi har här valt att alltid göra det vänstra subträdet en steg högre än det högra (förutom F_0 och F_1). Det betyder att det F_h kan konstrueras med hjälp av F_{h-1} och F_{h-2} enligt:



Träden kallas av naturliga skäl *Fibonacciträd*

Låt f_h beteckna antalet noder i F_h. Vi kan då uttrycka antalet noder med differensekvationen

$$f_h = 1 + f_{h-1} + f_{h-2} \tag{1}$$

Låt n_h vara antalet noder i ett *godtyckligt* AVL-träd med h nivåer. Då gäller naturligtvis

$$n_h \geq f_h \tag{2}$$

Om vi kan uttrycka f_h explicit så ger detta en relation mellan antalet nivåer h (dvs *höjden*) och antalet noder i ett AVL-träd. (Om vi t ex visste att $f_h = 2^h$ så skulle det innebära att $n_h \geq 2^h$ dvs $h \leq \log n_h$. Riktigt så enkelt och bra är det dock inte...)

Om man inte känner lösningen till ekvationen (1) eller inte behärskar teknik för att lösa den kan man ändå komma fram till det väsentliga resultatet genom en ganska enkel *uppskattning*.

Om vi skall kunna använda uppskattningen av f_h i relationen (2) så måste uppskattningen göras *nedåt* dvs vi skall skaffa en *undre gräns* för f_h.

Eftersom f_h är växande (fler nivåer kräver naturligtvis fler noder) så gäller

$$\begin{aligned} f_h &= 1 + f_{h-1} + f_{h-2} \geq \\ &\geq 1 + f_{h-2} + f_{h-2} \end{aligned}$$

dvs

$$f_h \geq 1 + 2f_{h-2} \tag{3}$$

Denna relation kan lösas genom "teleskopering" dvs genom att upprepa den ett lämpligt antal gånger:

$$\begin{aligned} f_h &\geq 1 + 2f_{h-2} \geq 1 + 2(1 + 2f_{h-4}) = \\ &= 1 + 2 + 4f_{h-4} \geq 1 + 2 + 4(1 + 2f_{h-6}) = \\ &= 1 + 2 + 4 + 8f_{h-6} \geq 1 + 2 + 4 + 8(1 + f_{h-8}) = \\ &\cdots \\ &= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k f_{h-2k} = \\ &= 2^k - 1 + 2^k f_{h-2k} \end{aligned}$$

Välj k sådant att $h - 2k = 1$ dvs $k = (h-1)/2$. Eftersom $f_1 = 1$ får vi då

$$\begin{aligned} n_h \geq f_h \geq 2^k - 1 + 2^k f_1 = 2^{k+1} - 1 = 2^{\frac{h+1}{2}} \\ \log_2(n_h + 1) \geq \frac{h+1}{2} \end{aligned}$$

$$h \leq 2\log_2(n_h + 1) - 1 \tag{4}$$

Slutsats: Ett AVL-träd kan inte bli mer än dubbelt så högt som det lägsta möjliga binära trädet.

Därav följer att *sökning* i ett AVL-träd med n noder tar garanterat $O(\log n)$ tid.

Eftersom in- och uttagsalgoritmerna bara rör noder längs vägen från roten till den nyinlagda roten kräver även högst dessa $O(\log n)$ tid.

Genom att lösa ekvationen (1) exakt så kan uppskattningen (4) skärpas till

$$h \leq 1.44\log_2(n_h + 2) + O(1) \tag{5}$$

Detta beskriver alltså hur många nodbesök man behöver göra i *värsta fall* i det *värsta* AVL-trädet att jämföras med de $1.396\log_2 n$ försök man behöver göra i *genomsnitt* i det *genomsnittliga* binära sökträdet.