Självbalanserande träd
AVL-träd

Koffman & Wolfgang

- kapitel 9, avsnitt 1–2

Balanserade träd

Nodbalanserat träd:

- skillnaden i antalet noder mellan vänster och höger delträd är högst 1

Höjdbalanserat träd:

- skillnaden i höjd (djup) mellan vänster och höger delträd är högst 1

Nivåbalanserat träd:

- alla nivåer är fyllda, utom den nedersta som fylls på från vänster till höger

Notera att höjden av alla dessa träd är logaritmisk:

- $h = O(\log n)$, där n är antalet noder i trädet

Självbalanserande sökträd

Effektiviteten hos ett binärt sökträd är proportionell mot höjden på trädet:

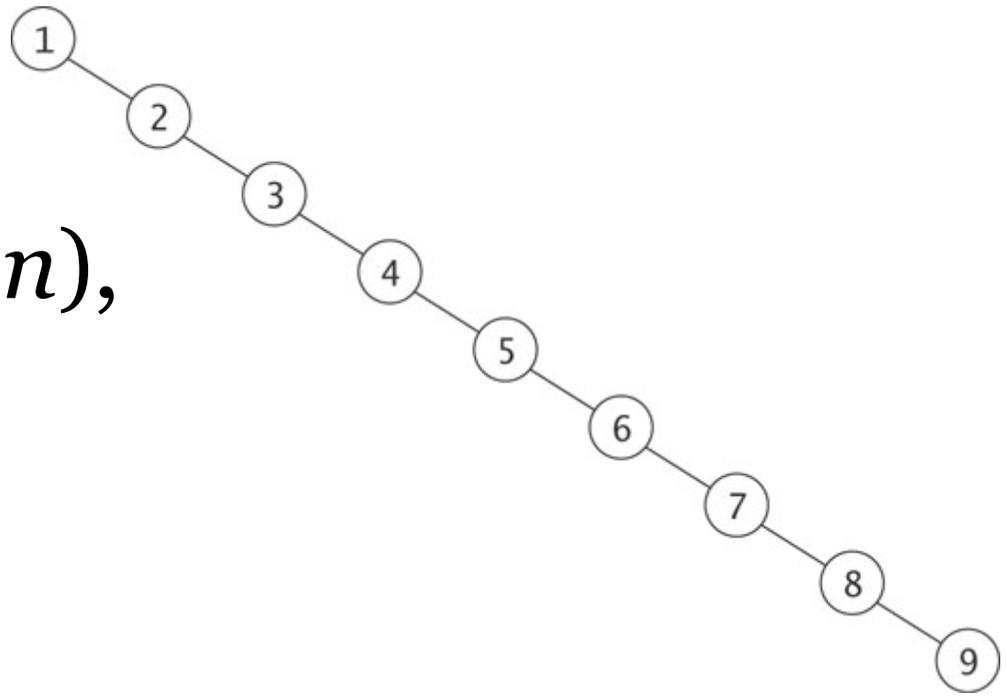
- ett balanserat träd är effektivare än ett obalanserat
- i värsta fall kan sökträdet ha linjär höjd, dvs $h = O(n)$
- då blir komplexiteten också linjär

För att lösa detta problem introducerar vi självbalanserande träd:

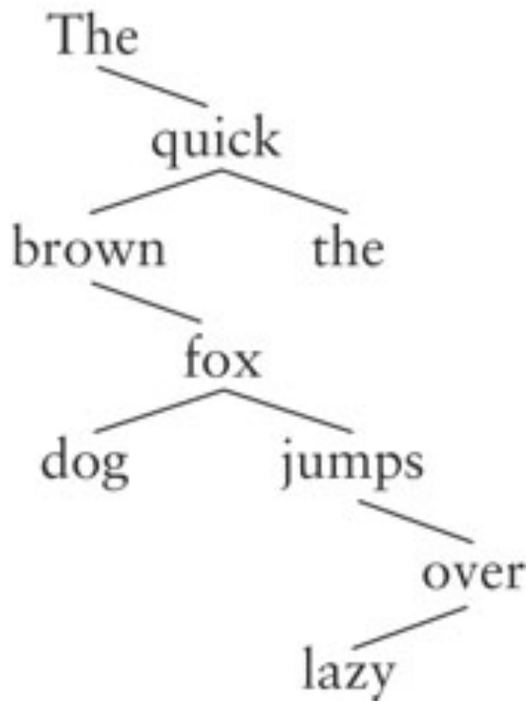
- vid insättning och uttagning så balanserar trädet om sig om det är nödvändigt

Obalanserade träd

Sökningar i detta obalanserade träd är O(n), inte O(log n)

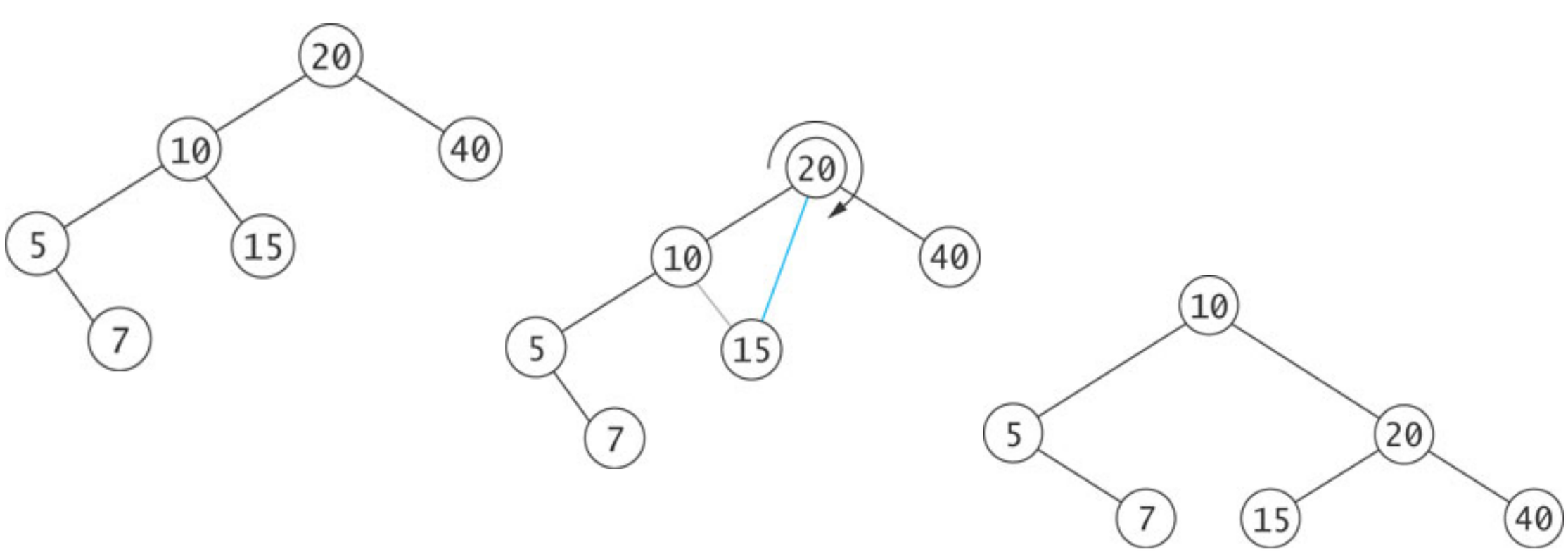


Här är ett mer realistiskt exempel på ett obalanserat träd

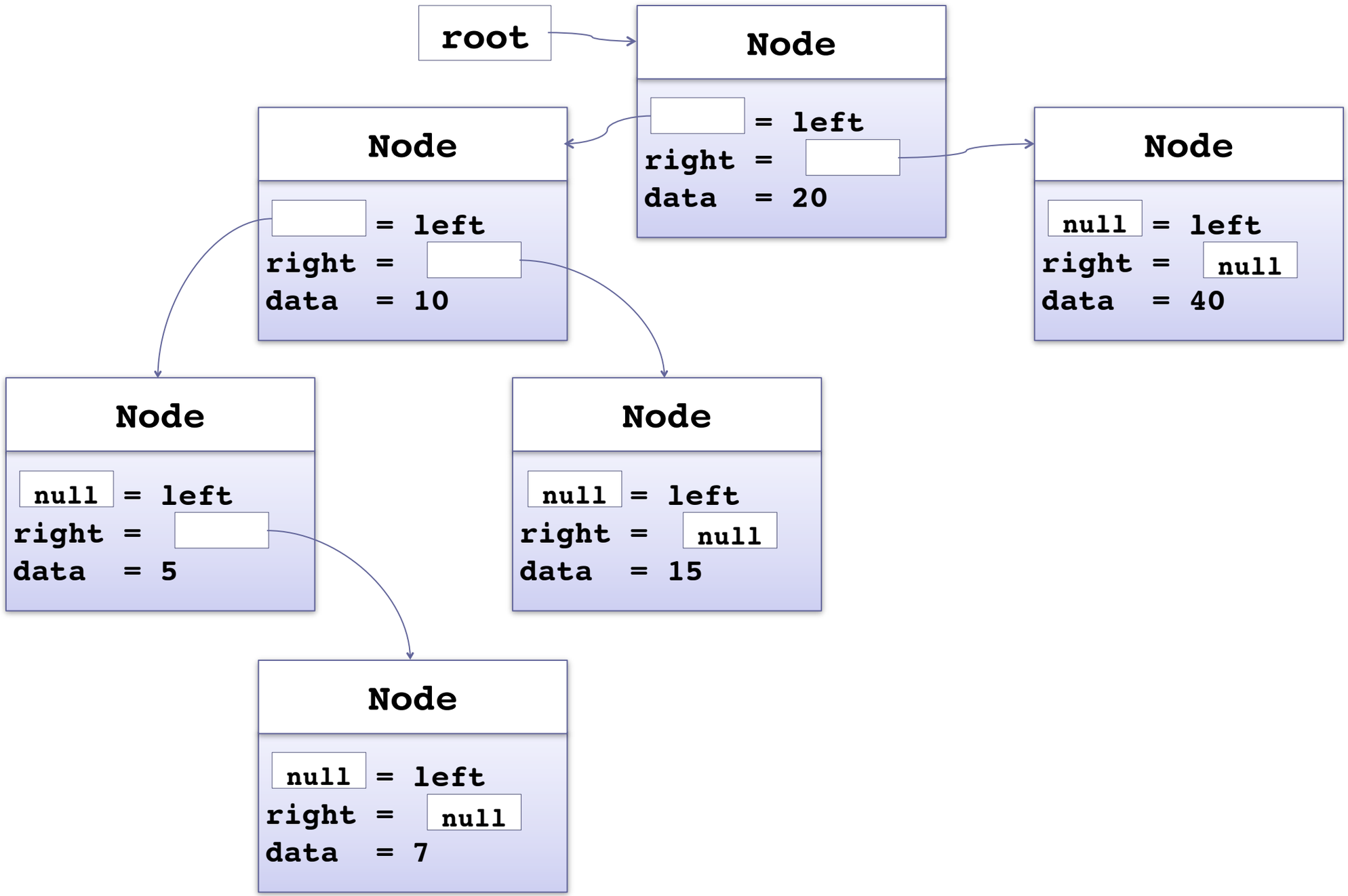


Rotering

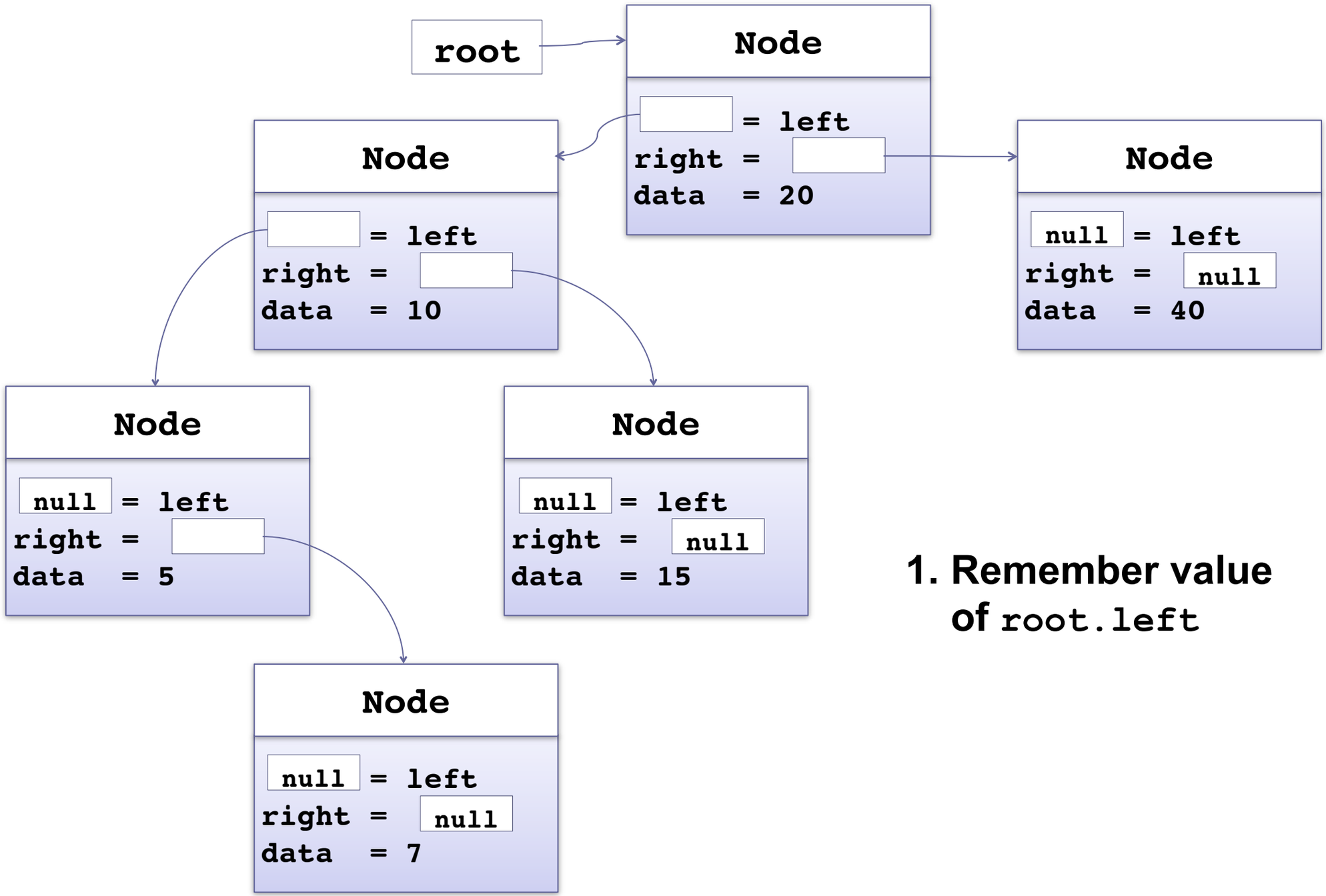
Vi behöver en operation på binära träd som ändrar de relativa höjderna mellan vänster och höger delträd, men bibehåller sökträdsegenskapen



Rotering, exempel

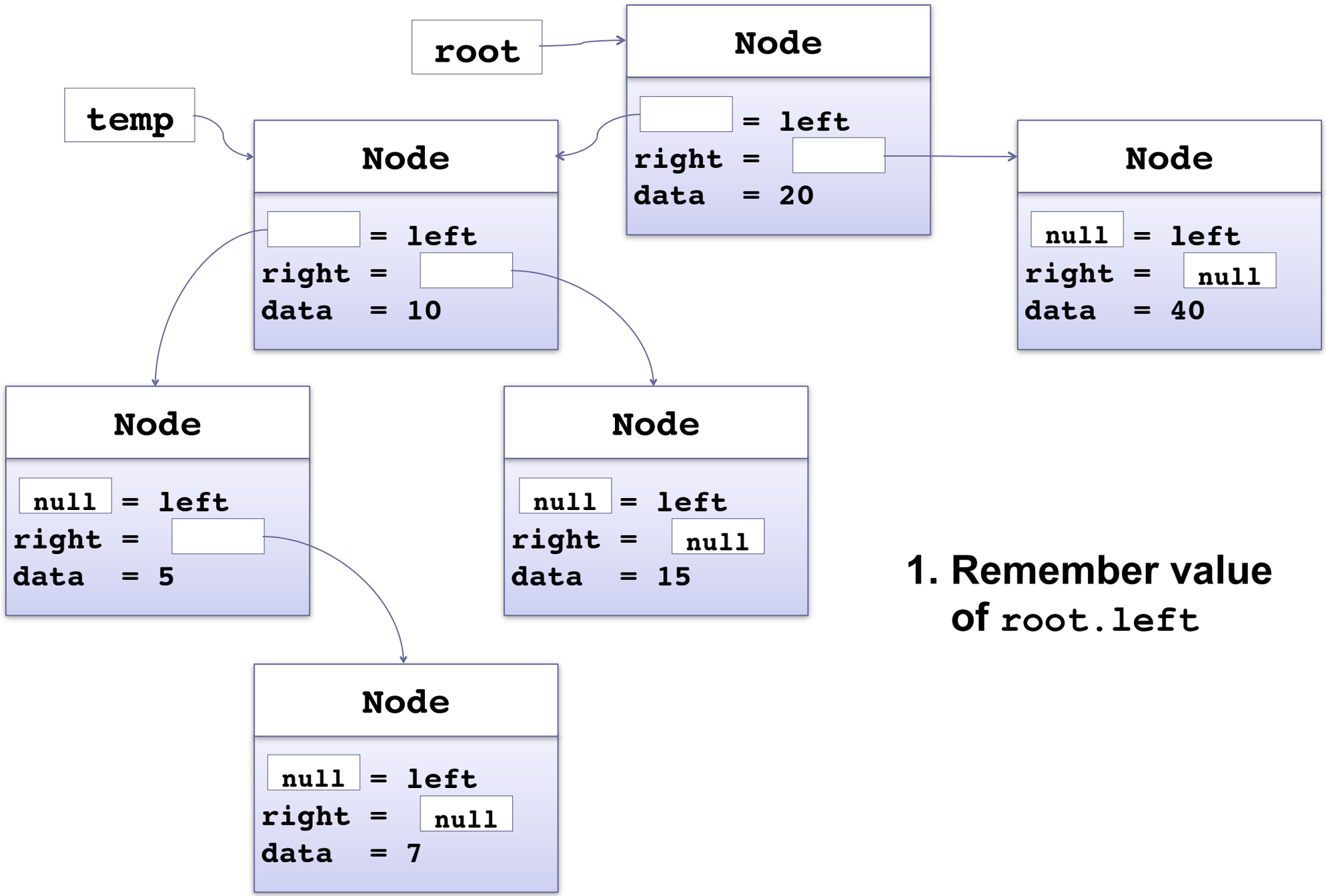


Rotering, exempel



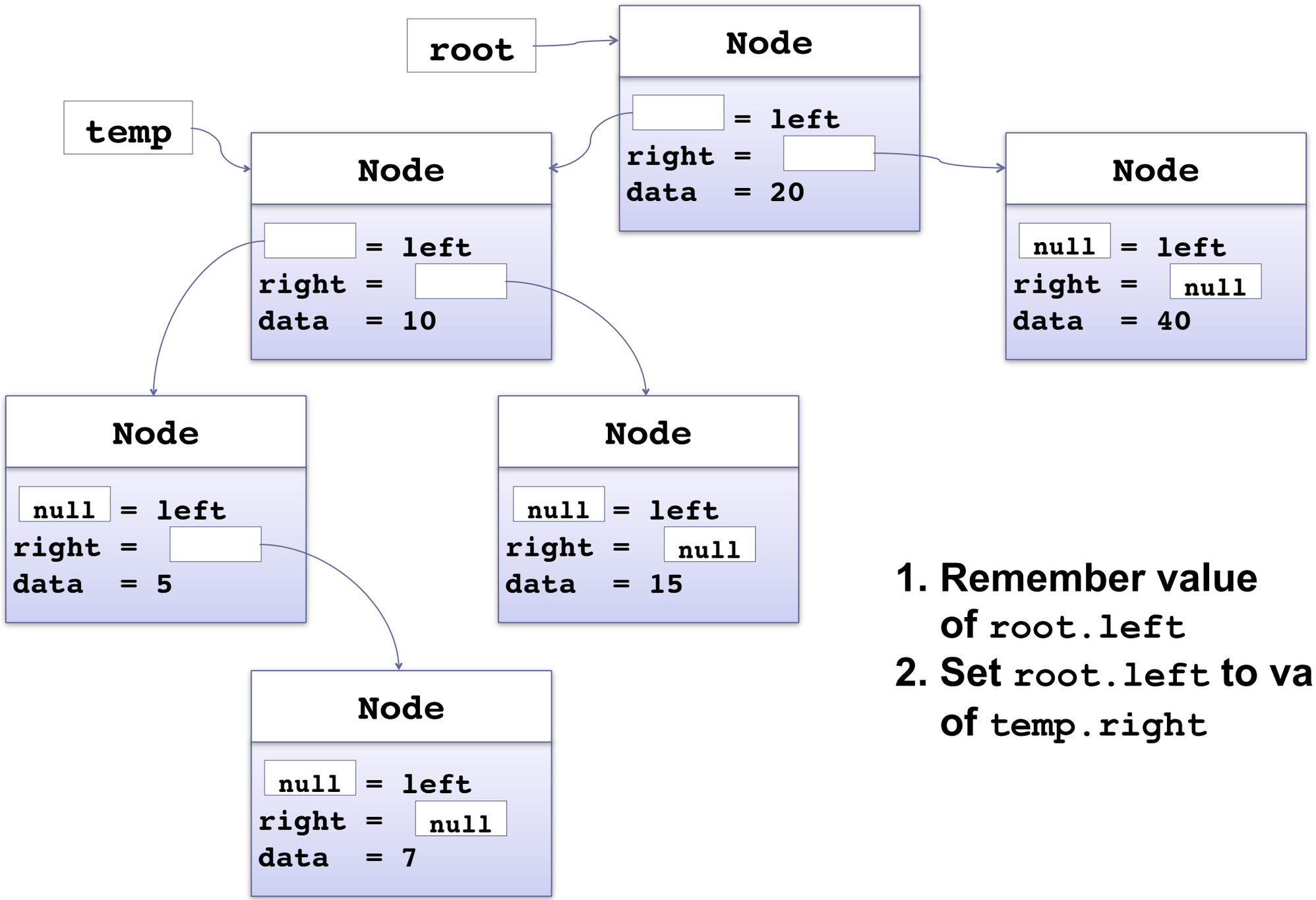
1. Remember value of `root.left`

Rotering, exempel



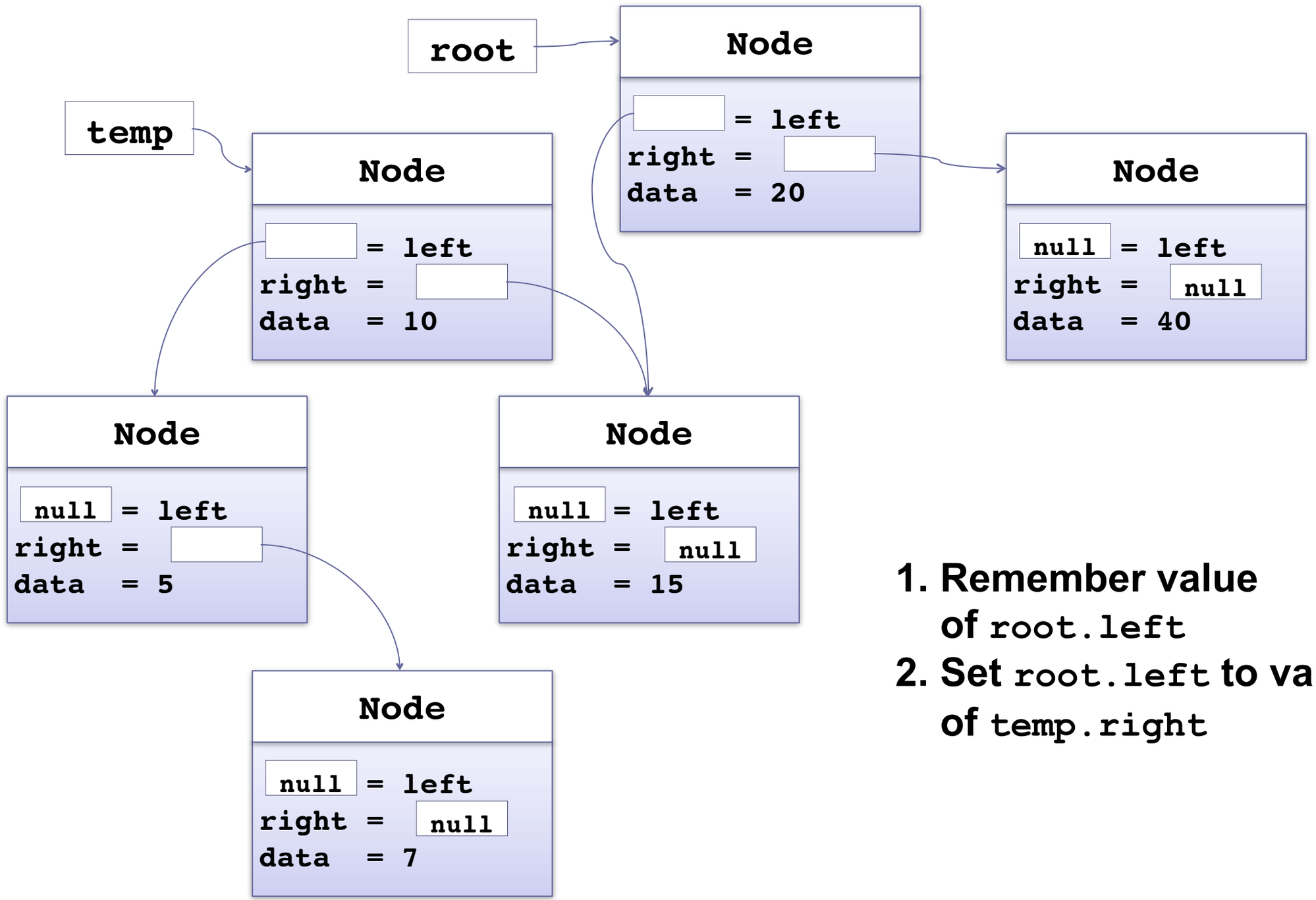
1. **Remember value of `root.left`**

Rotering, exempel



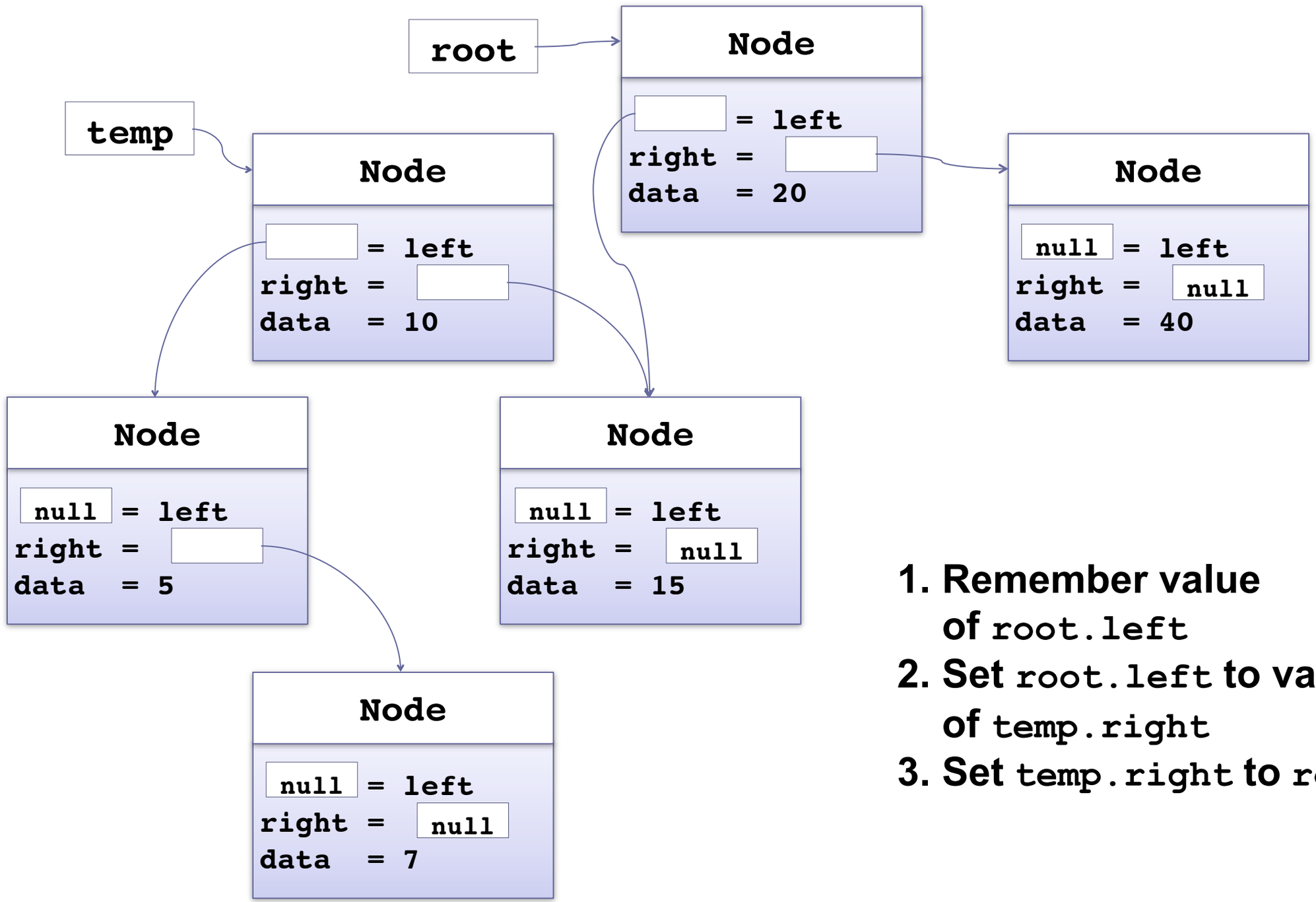
1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`

Rotering, exempel



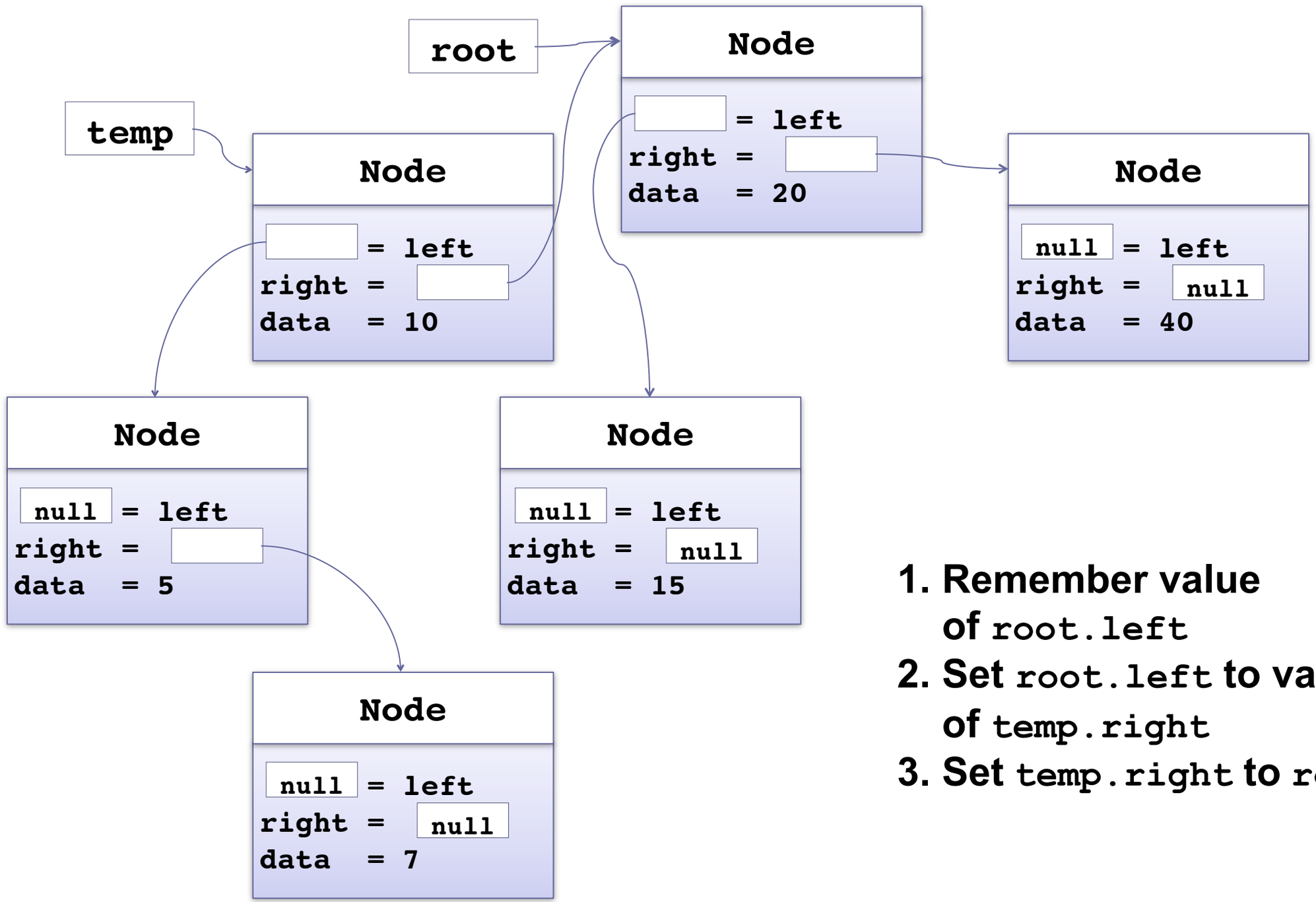
1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`

Rotering, exempel



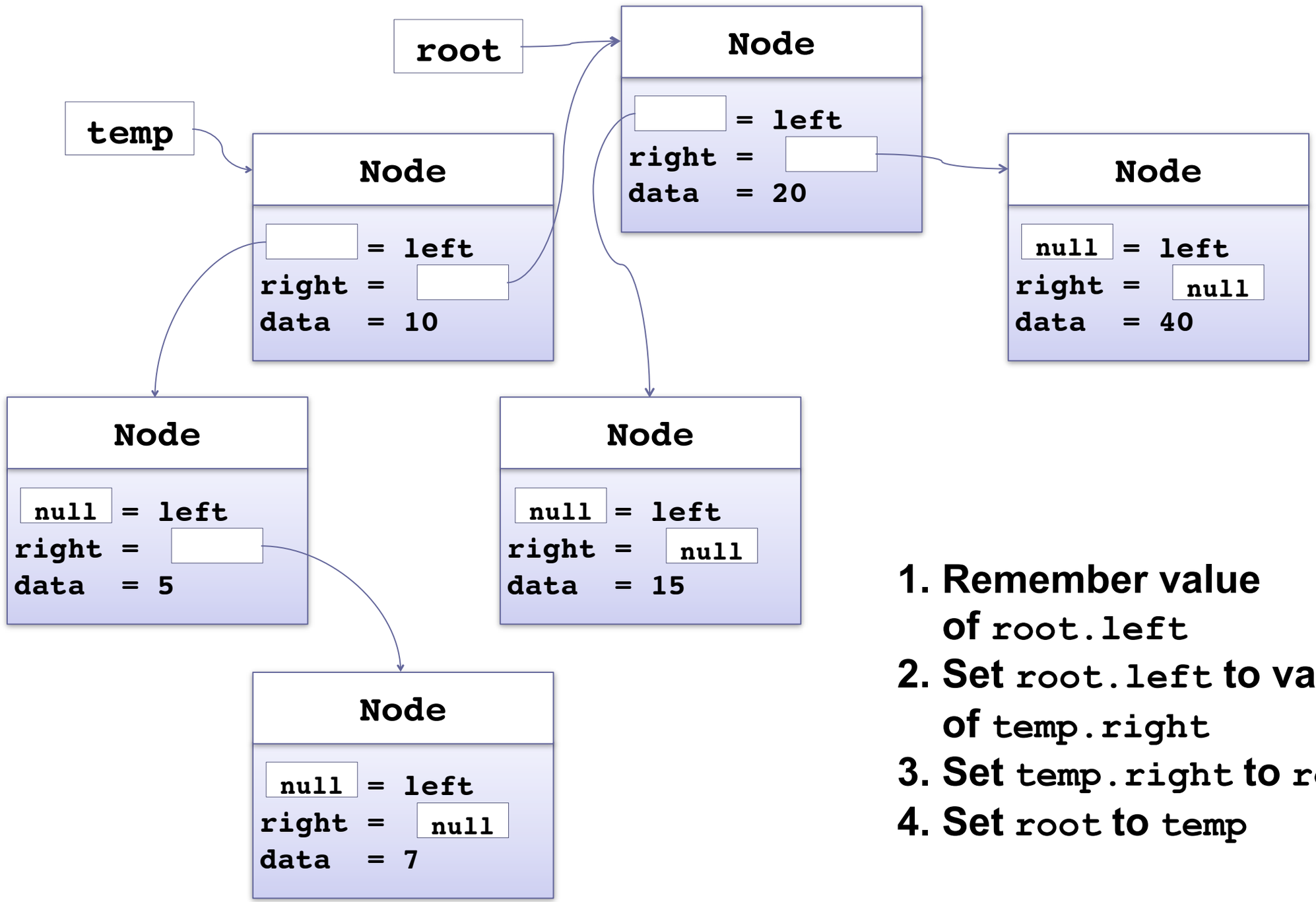
1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`

Rotering, exempel



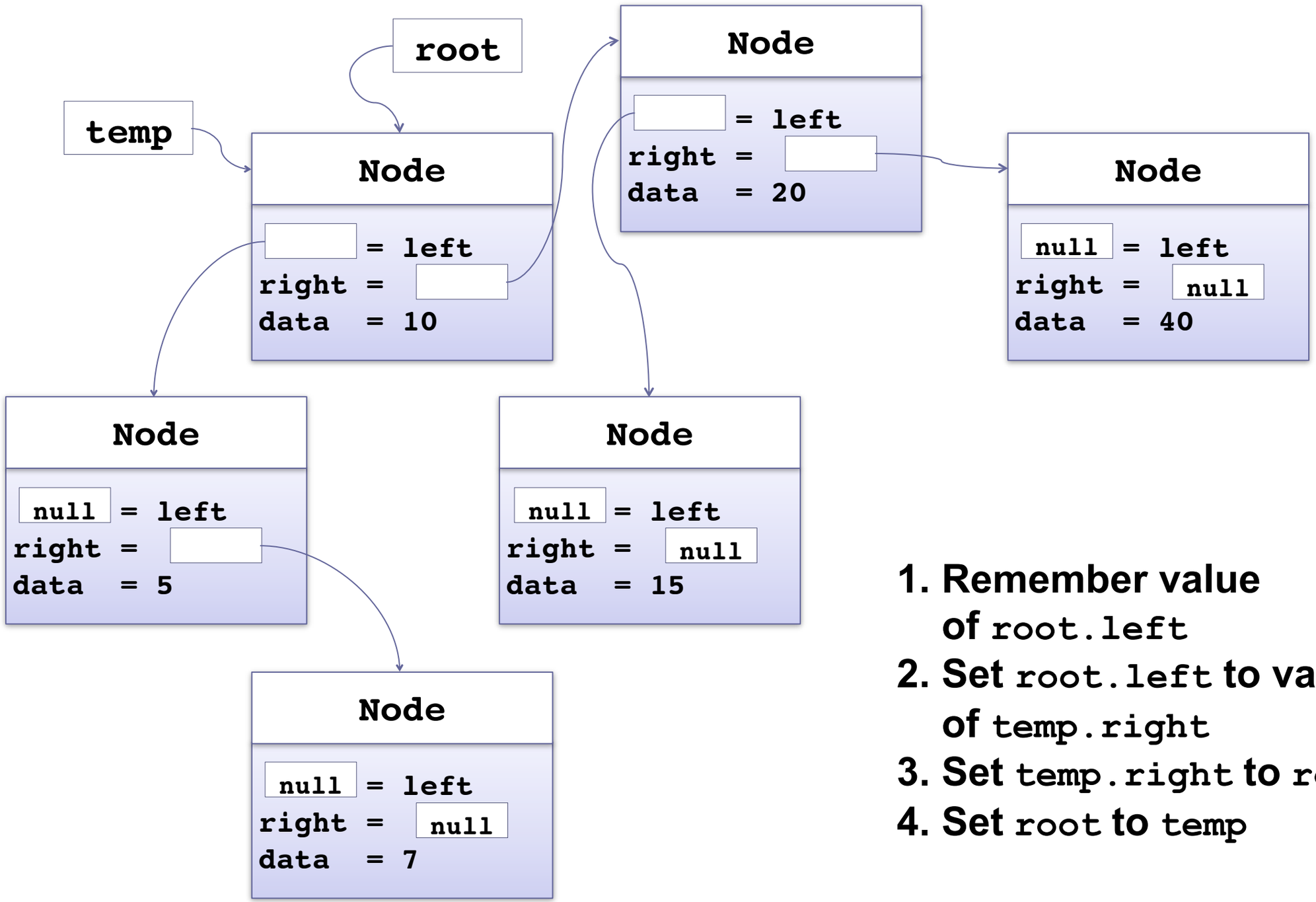
1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`

Rotering, exempel



1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`
4. Set `root` to `temp`

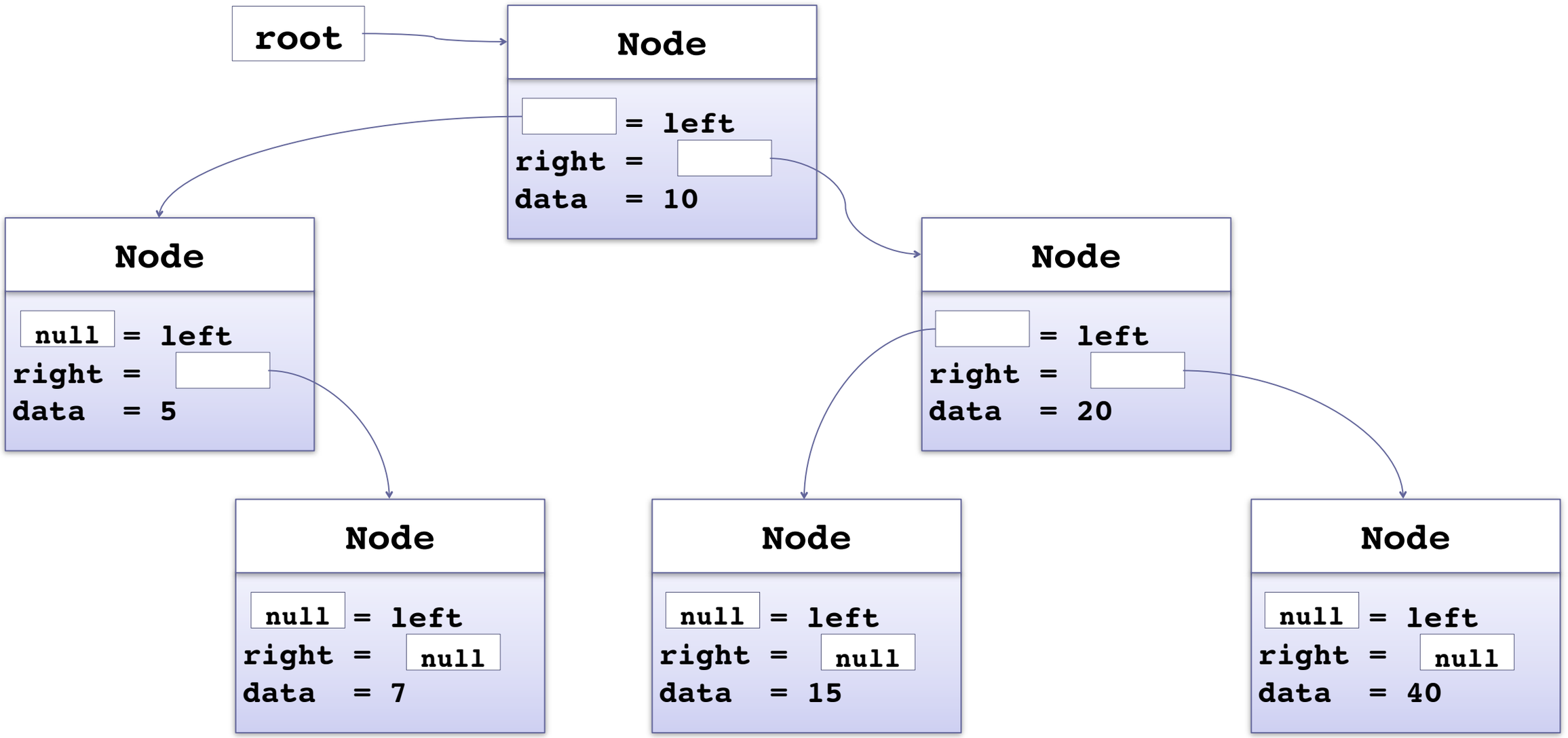
Rotering, exempel



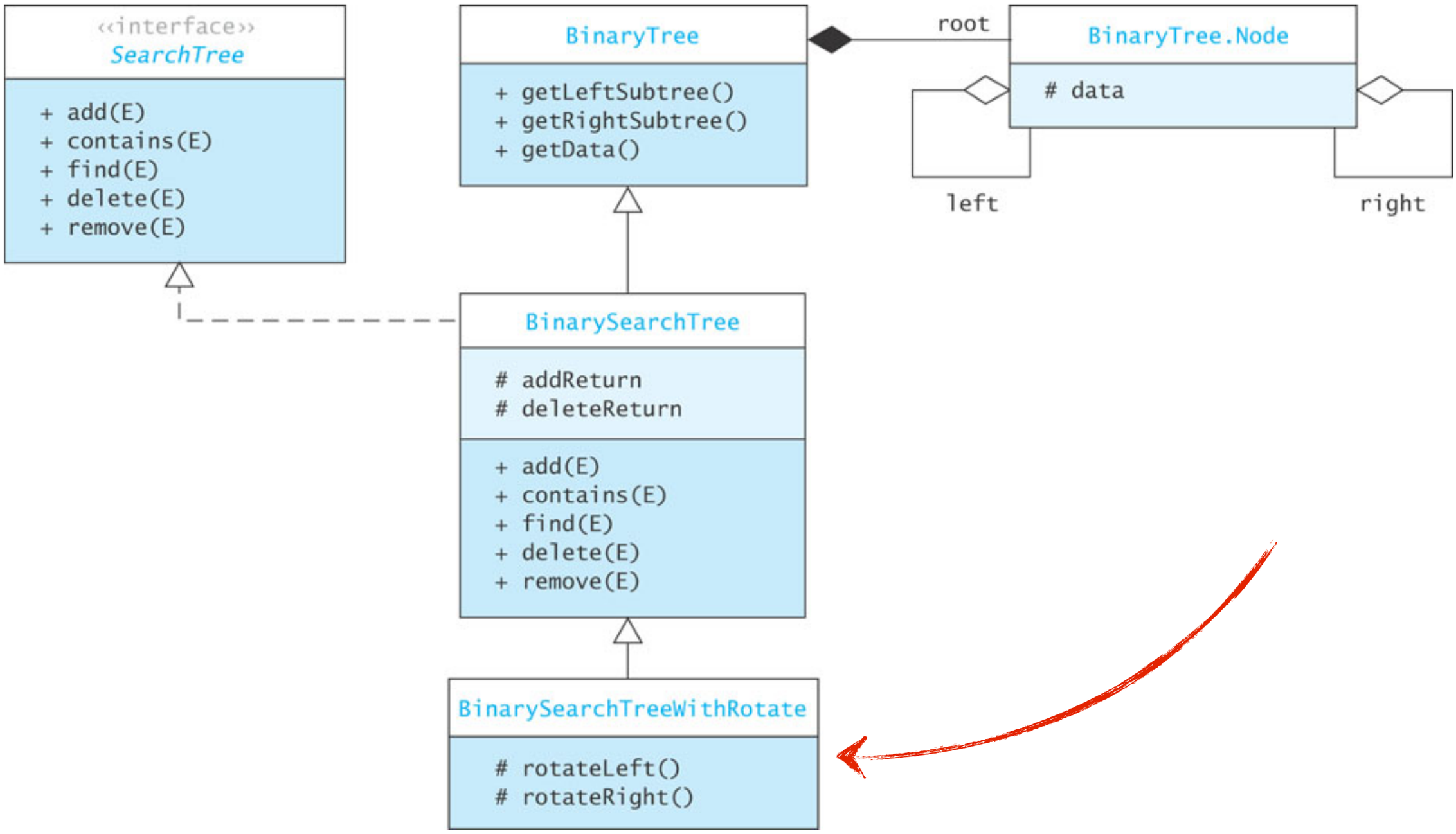
1. Remember value of `root.left`
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`
4. Set `root` to `temp`

Rotering, exempel

Rotering, exempel



class BinarySearchTreeWithRotate



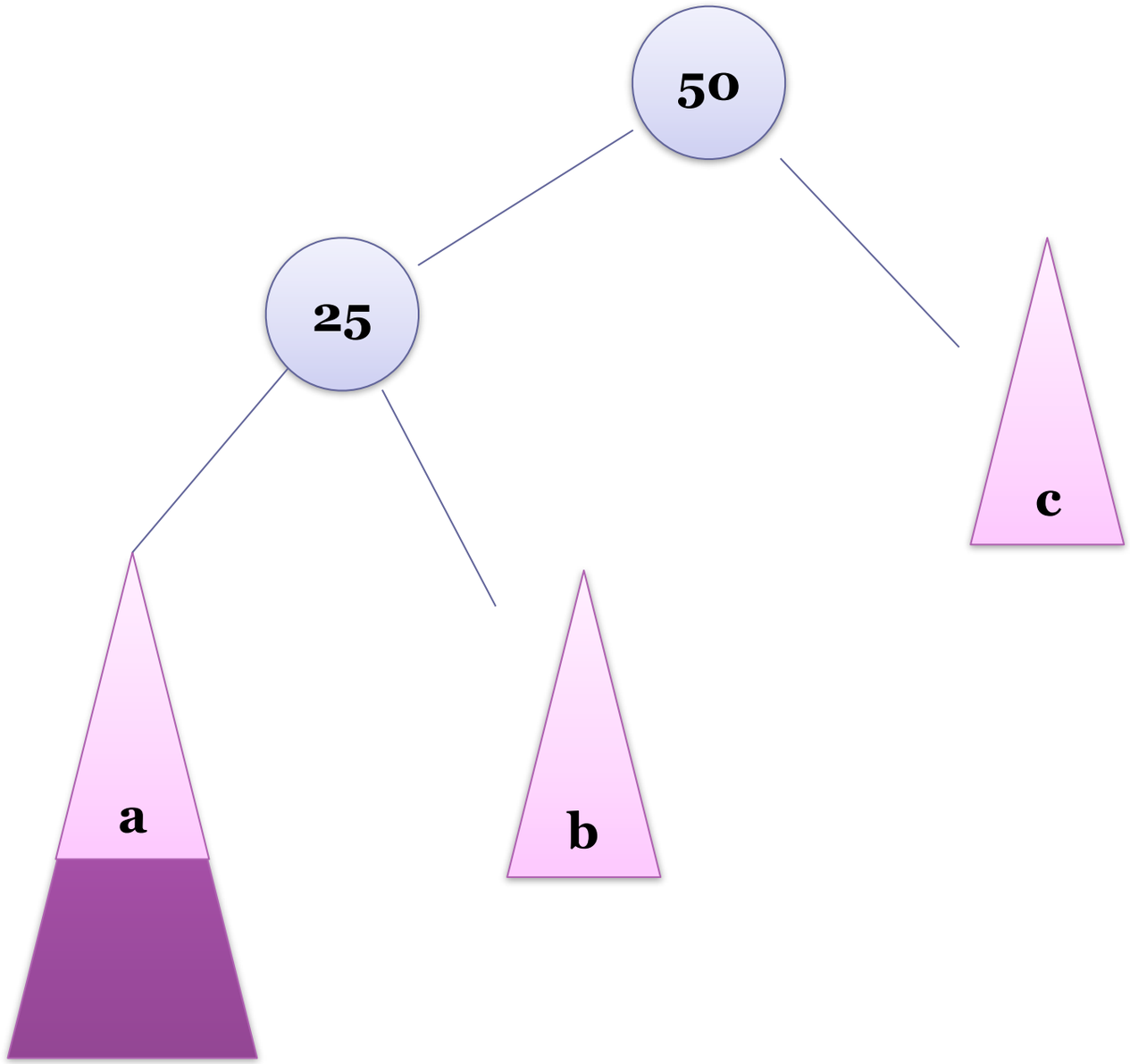
AVL-träd

AVL står för Adelson-Velskii (Адельсóн-Вéльский) och Landis (Лáндис) som kom på trädalgoritmerna

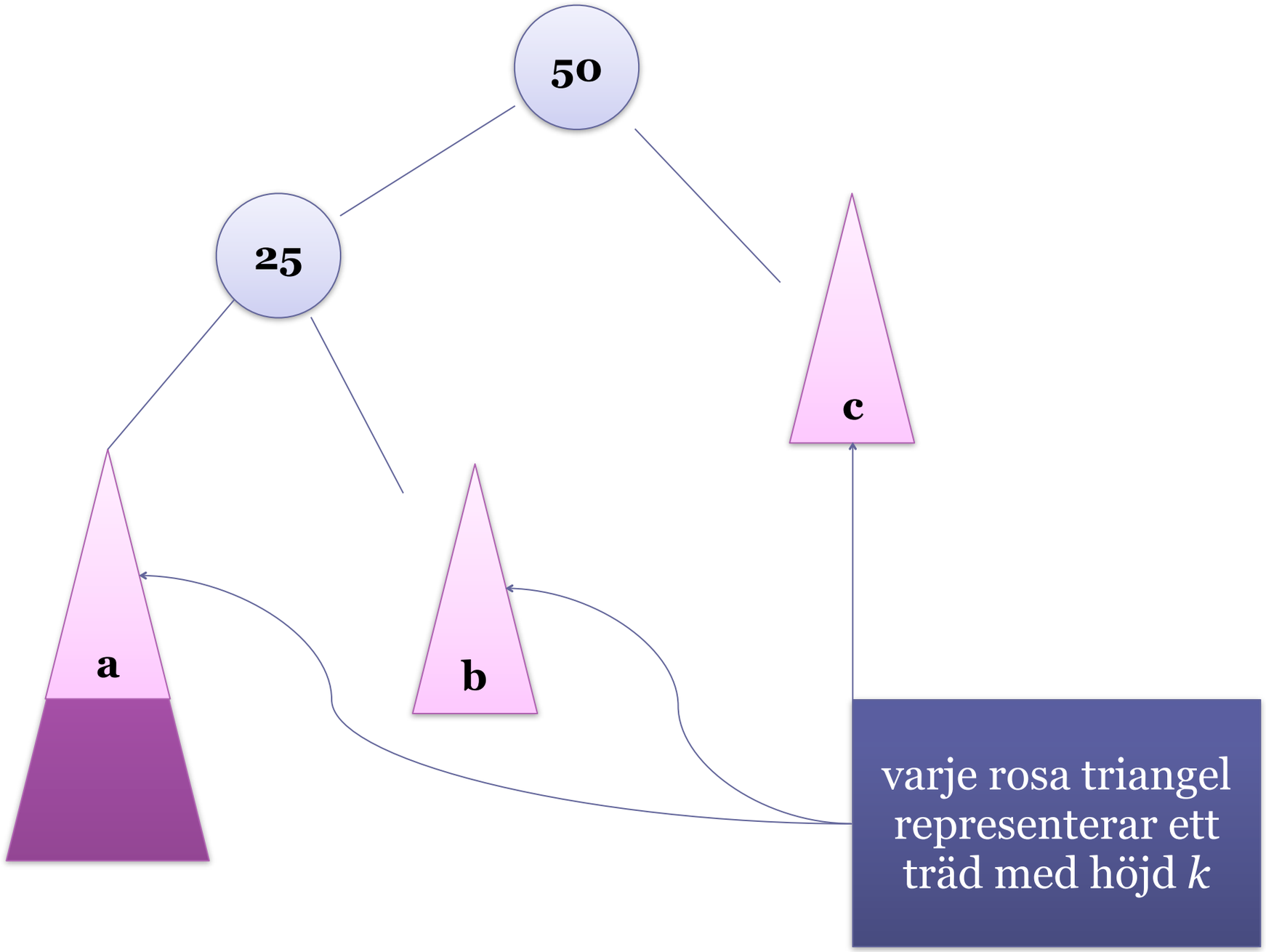
AVL-trädet håller reda på vänster-höger-balansen i varje delträd:

- när element läggs till eller tas bort från trädet, uppdateras balansen för alla delträd från insättningspunkten upp till roten
- om balansen hamnar utanför –1 till +1, roteras trädet för att få det i balans igen

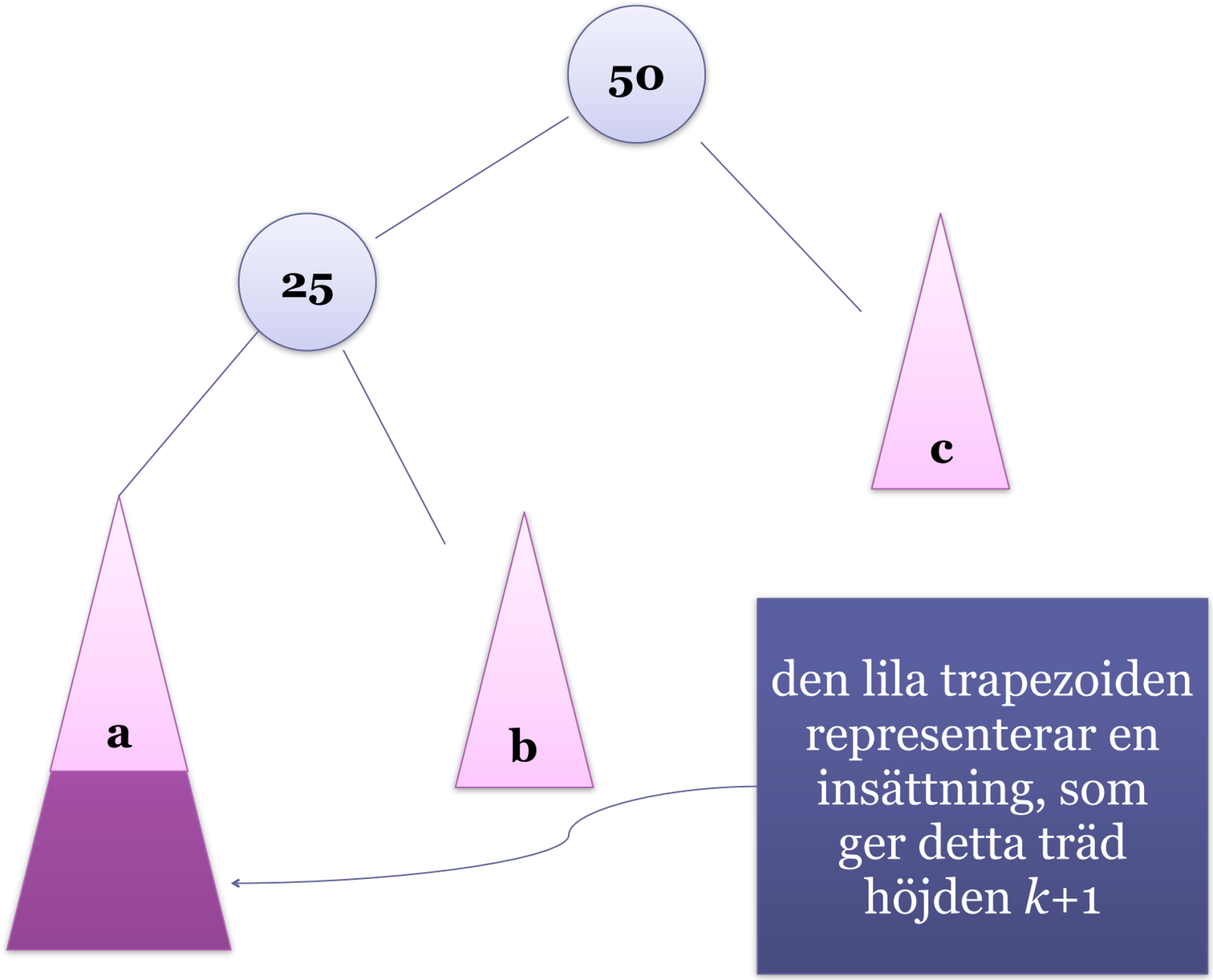
Balansera ett vänster-vänsterträd



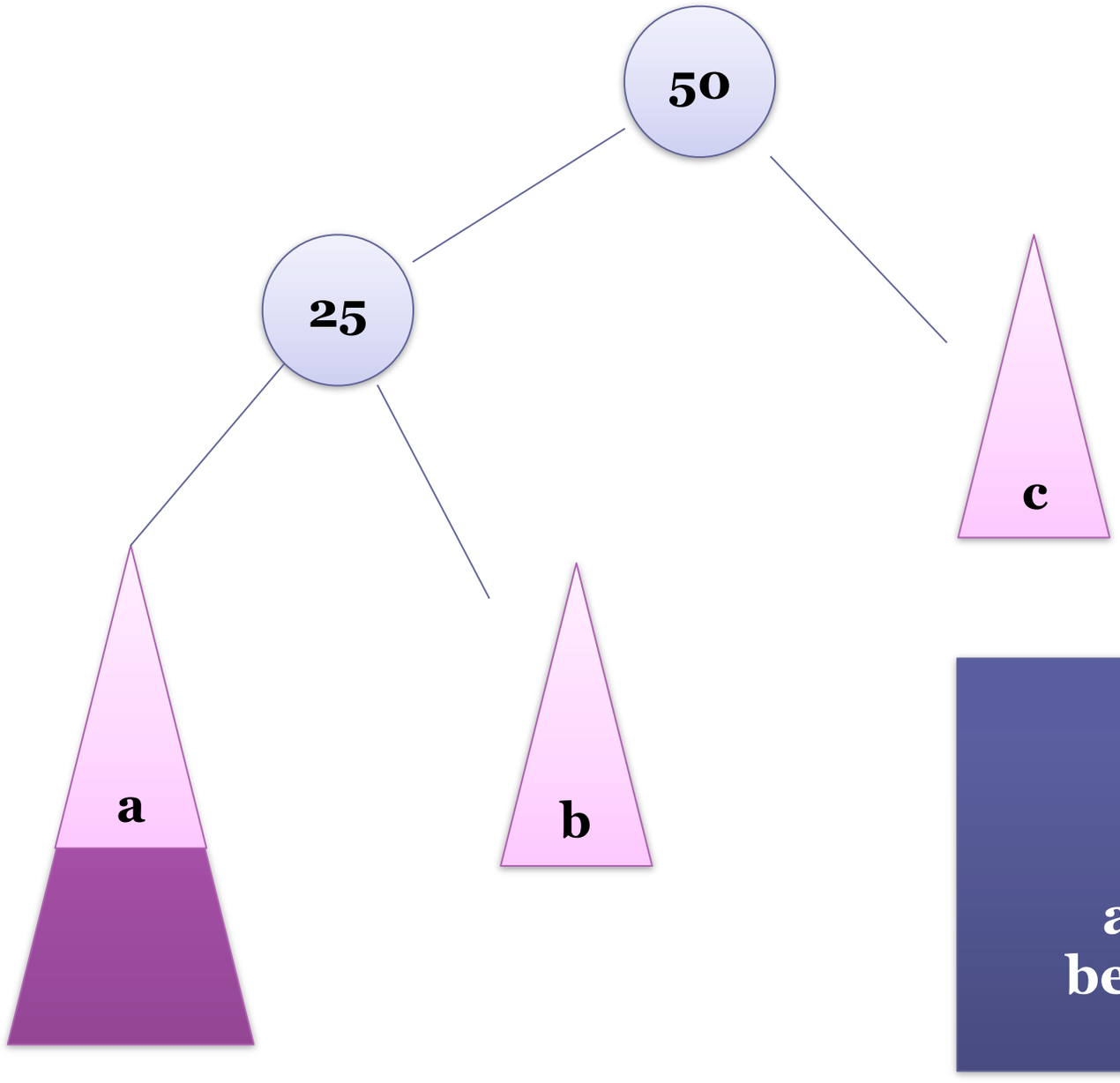
Balansera ett vänster-vänsterträd



Balansera ett vänster-vänsterträd

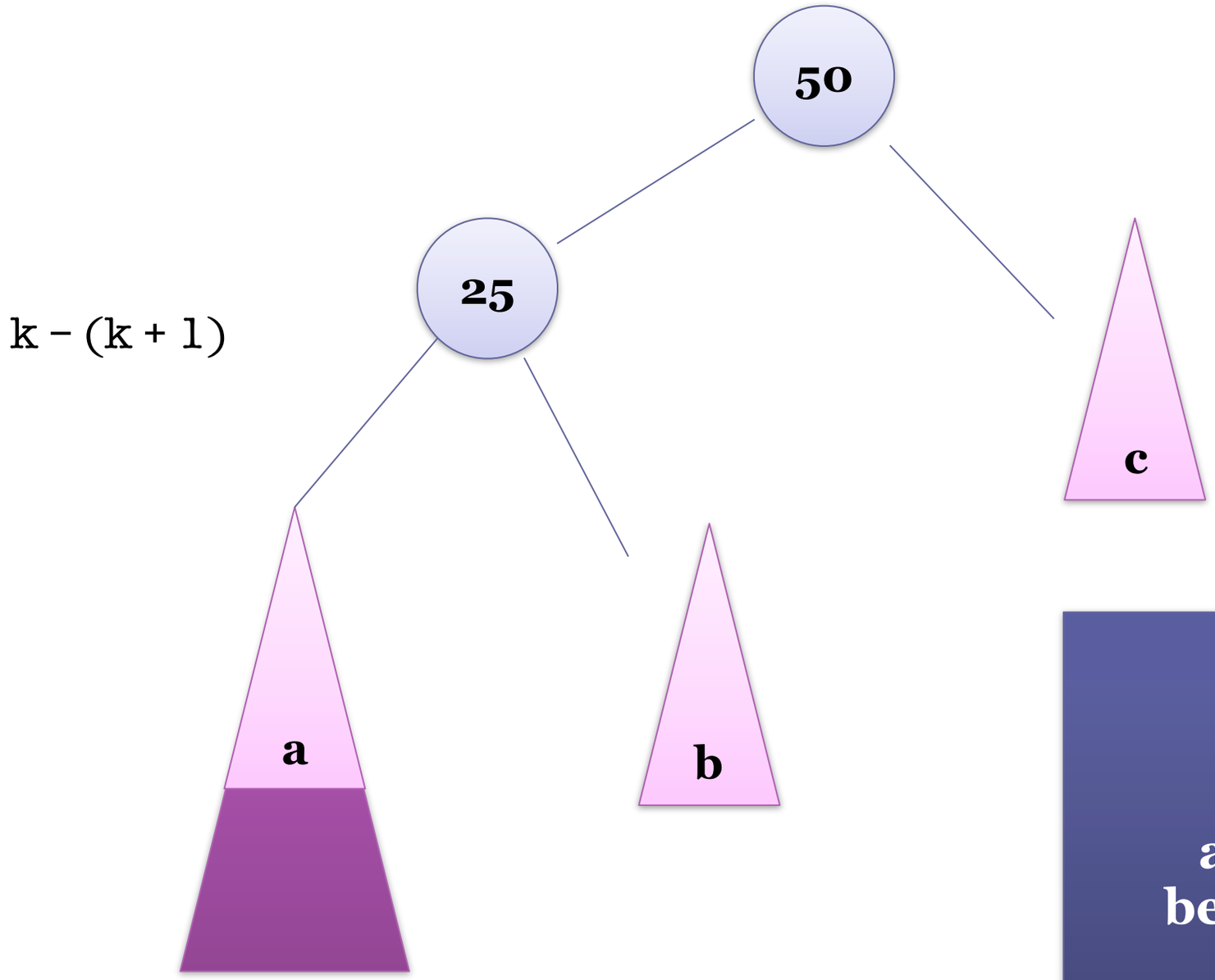


Balansera ett vänster-vänsterträd



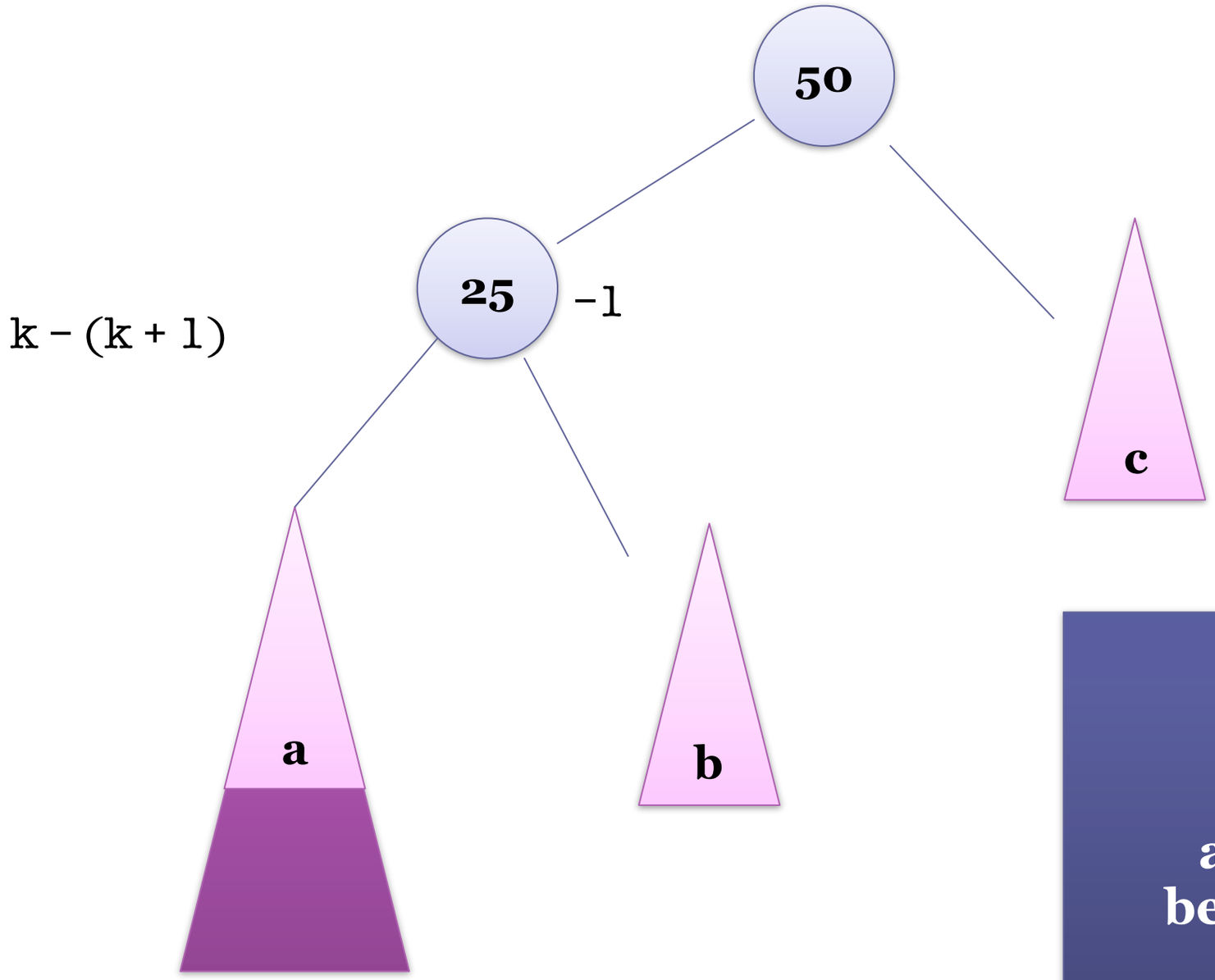
formeln
$$h_R - h_L$$
används för att beräkna balansen

Balansera ett vänster-vänsterträd



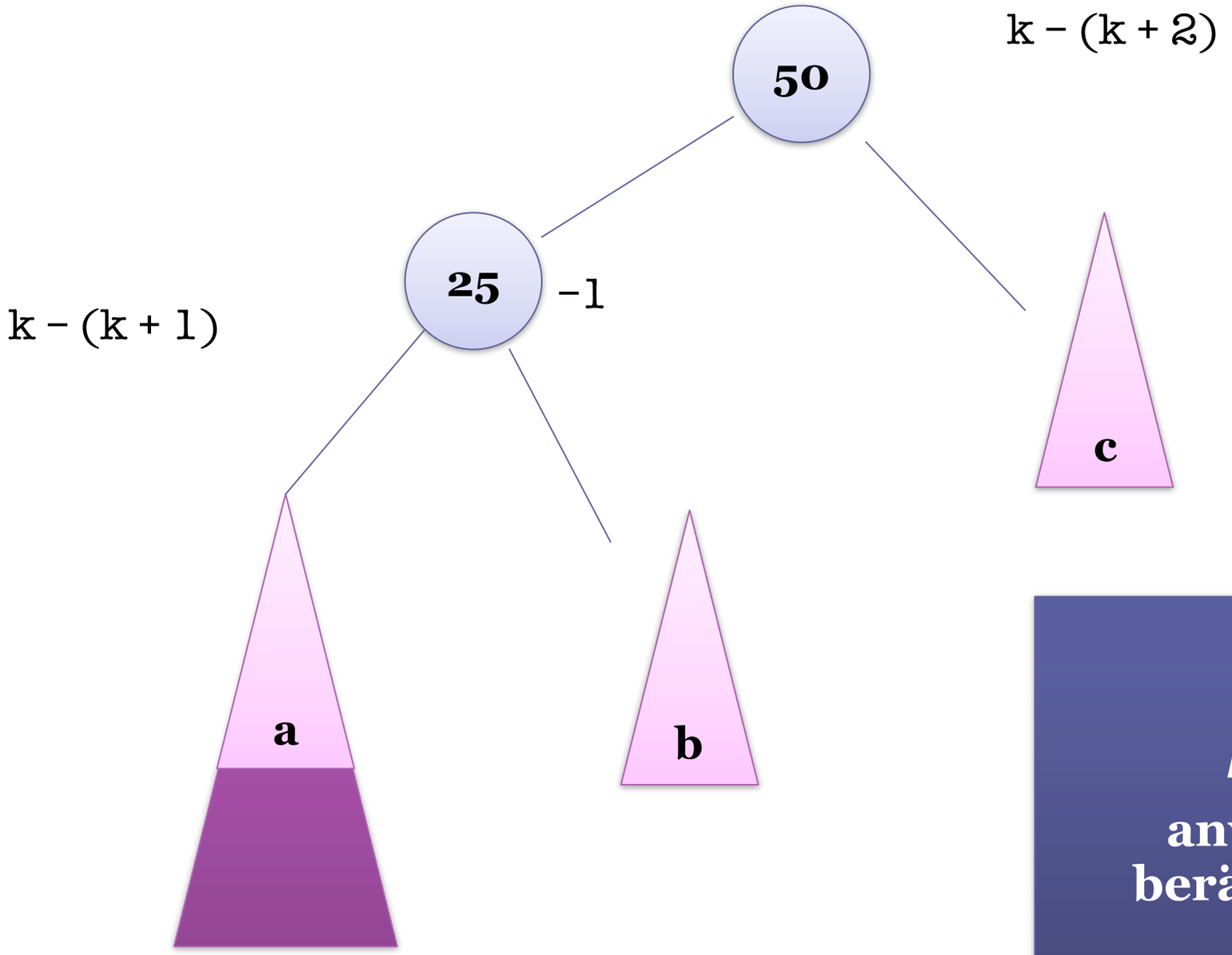
formeln
$h_R - h_L$
används för att beräkna balansen

Balansera ett vänster-vänsterträd



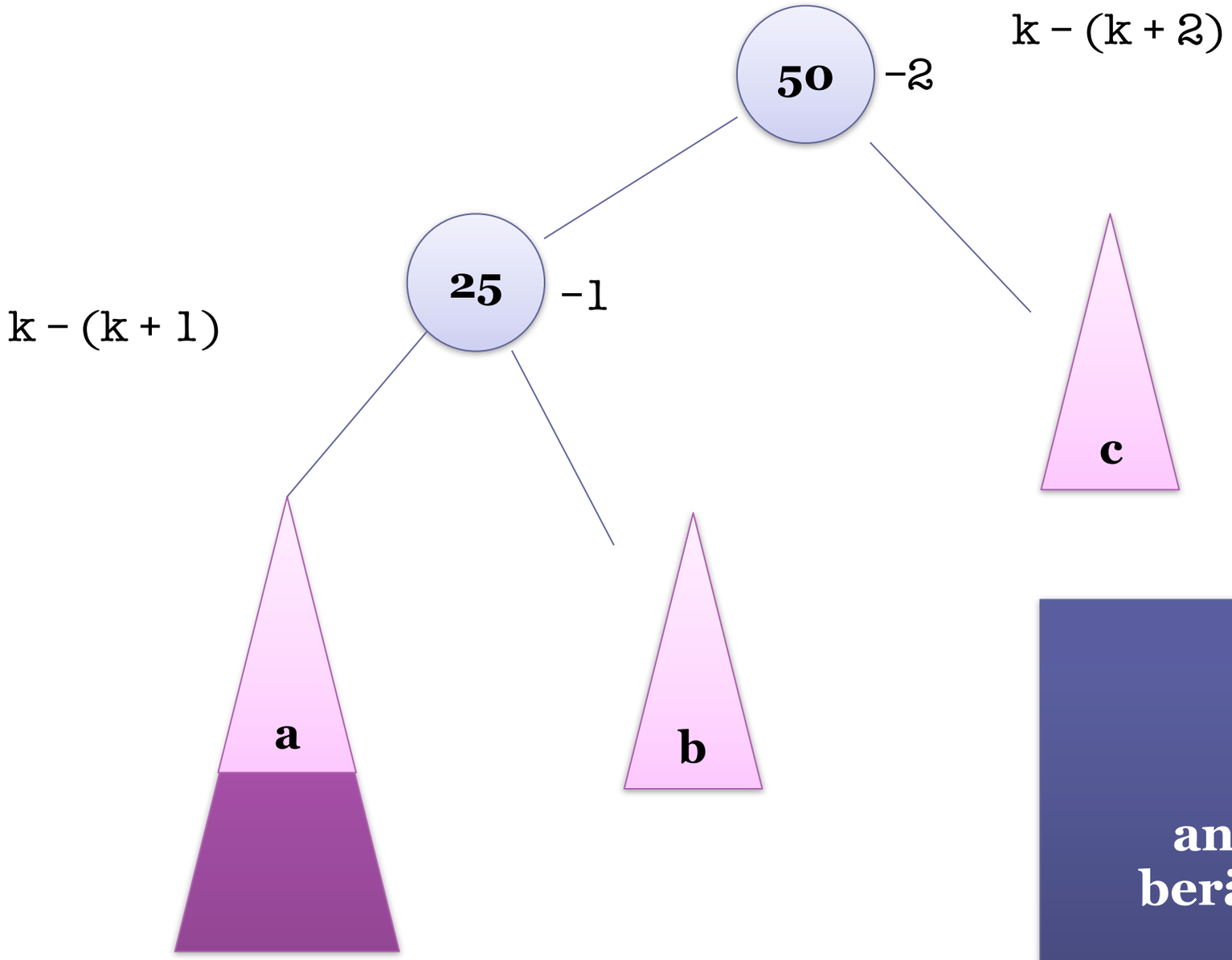
formeln
$h_R - h_L$
används för att beräkna balansen

Balansera ett vänster-vänsterträd



formeln
$h_R - h_L$
används för att beräkna balansen

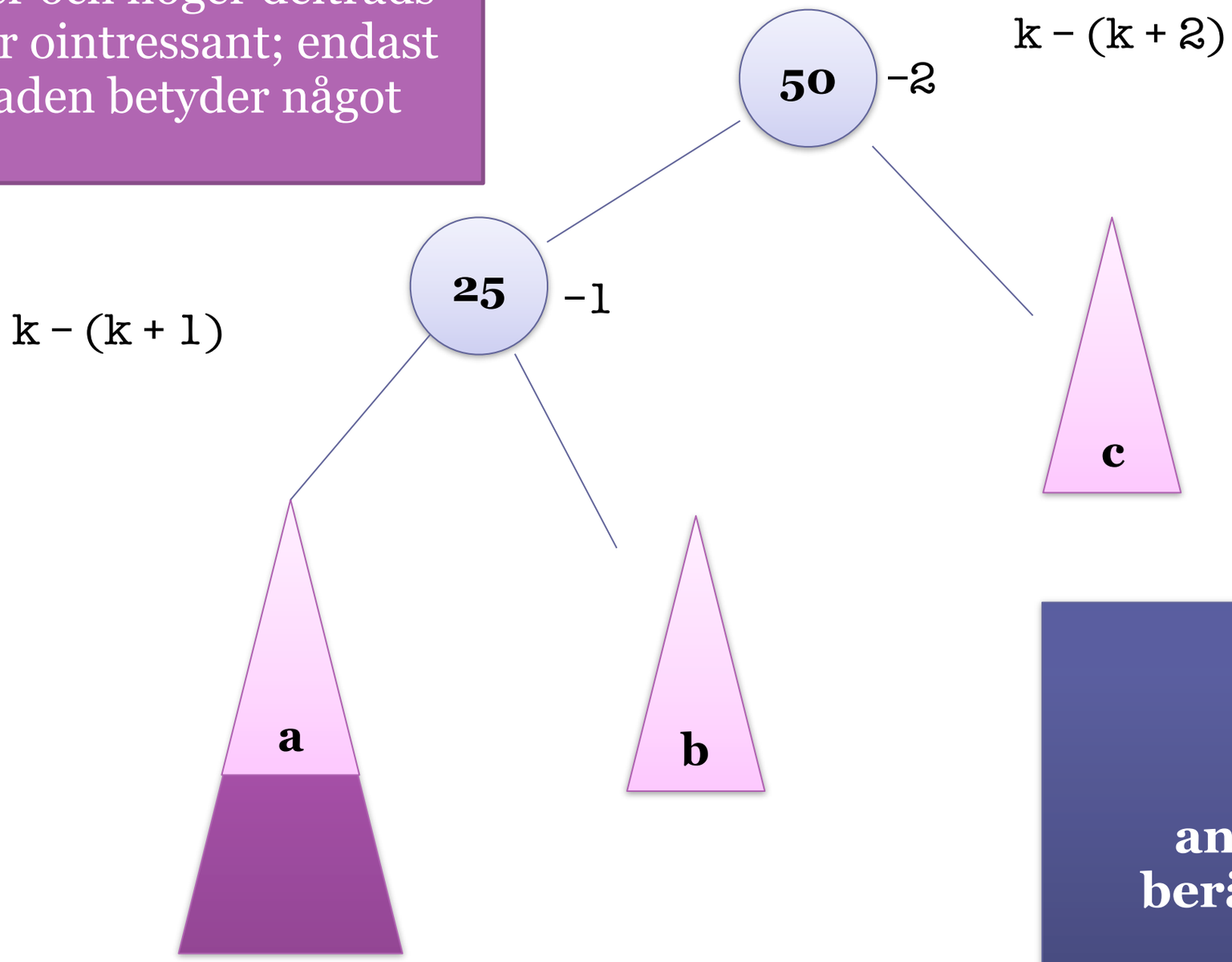
Balansera ett vänster-vänsterträd



formeln
$h_R - h_L$
används för att beräkna balansen

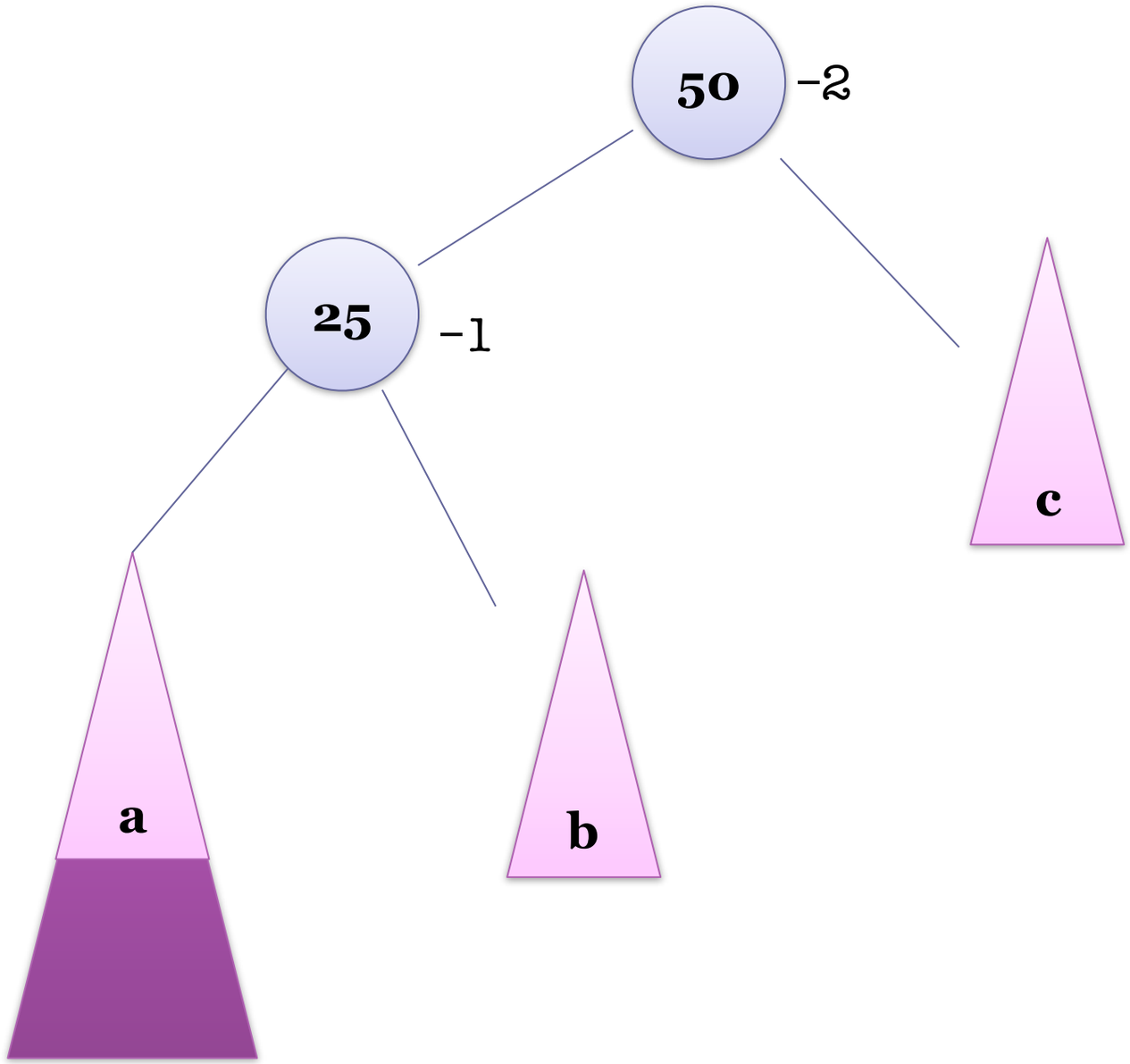
Balansera ett vänster-vänsterträd

vänster och höger delträds höjd är ointressant; endast skilladen betyder något



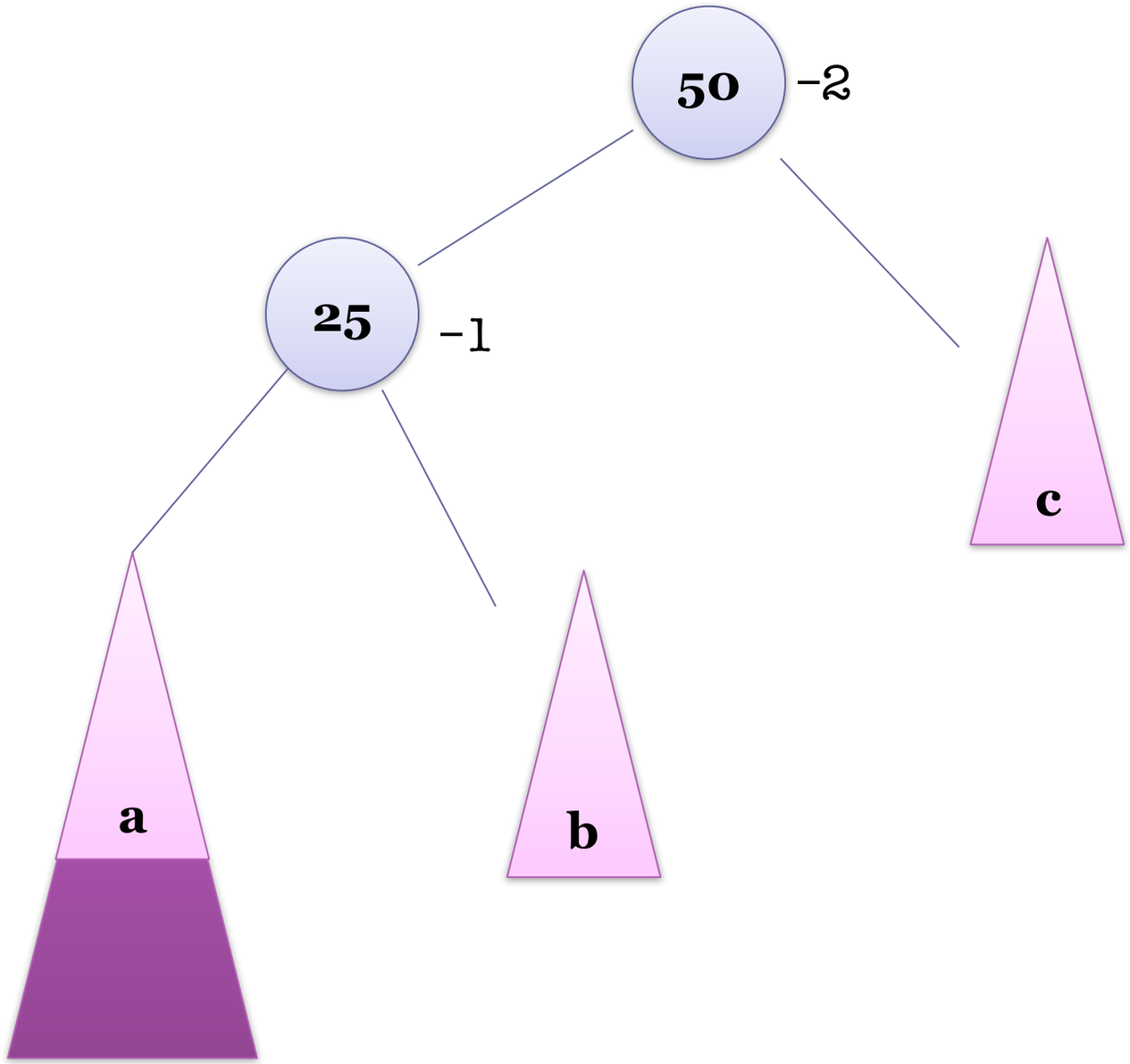
formeln
$h_R - h_L$
används för att beräkna balansen

Balansera ett vänster-vänsterträd



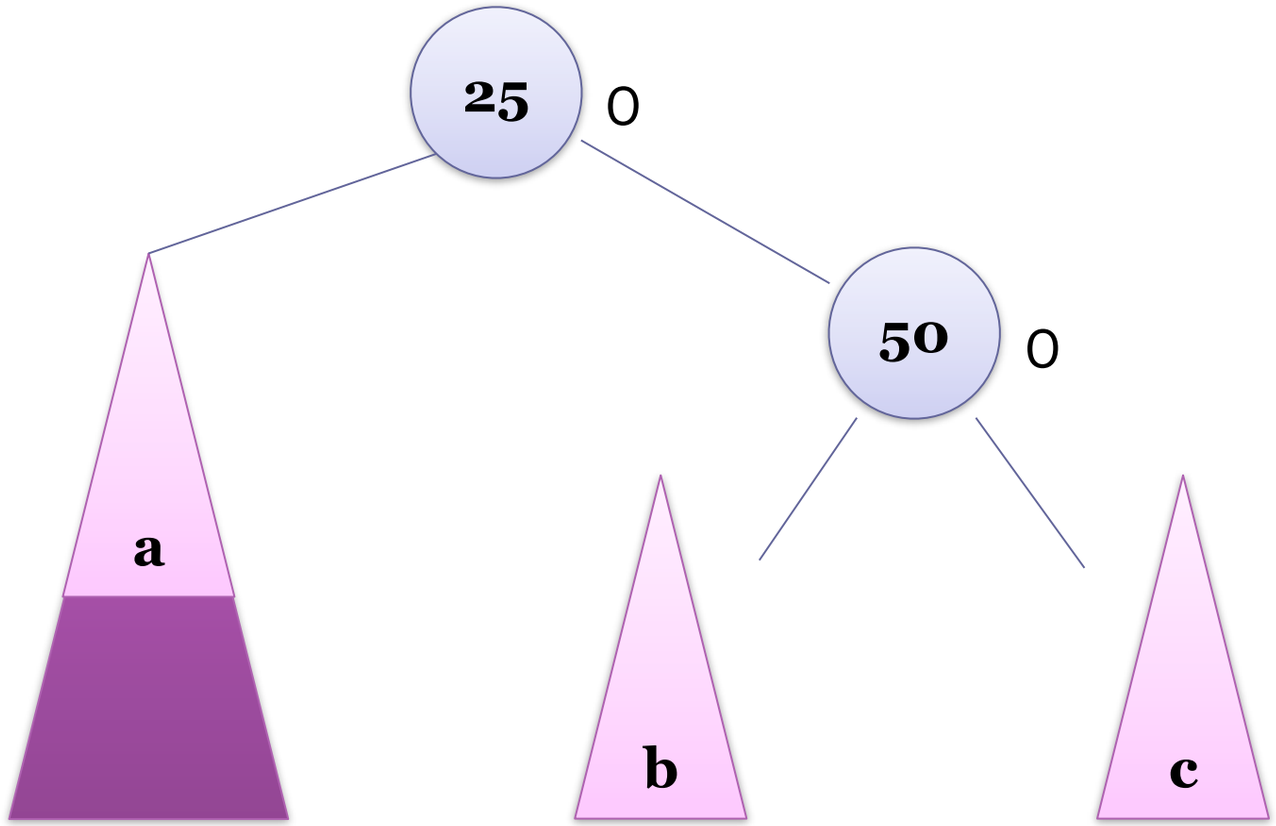
när både roten och vänster delträd ”lutar åt vänster”, så kallas trädet ett vänster-vänsterträd

Balansera ett vänster-vänsterträd

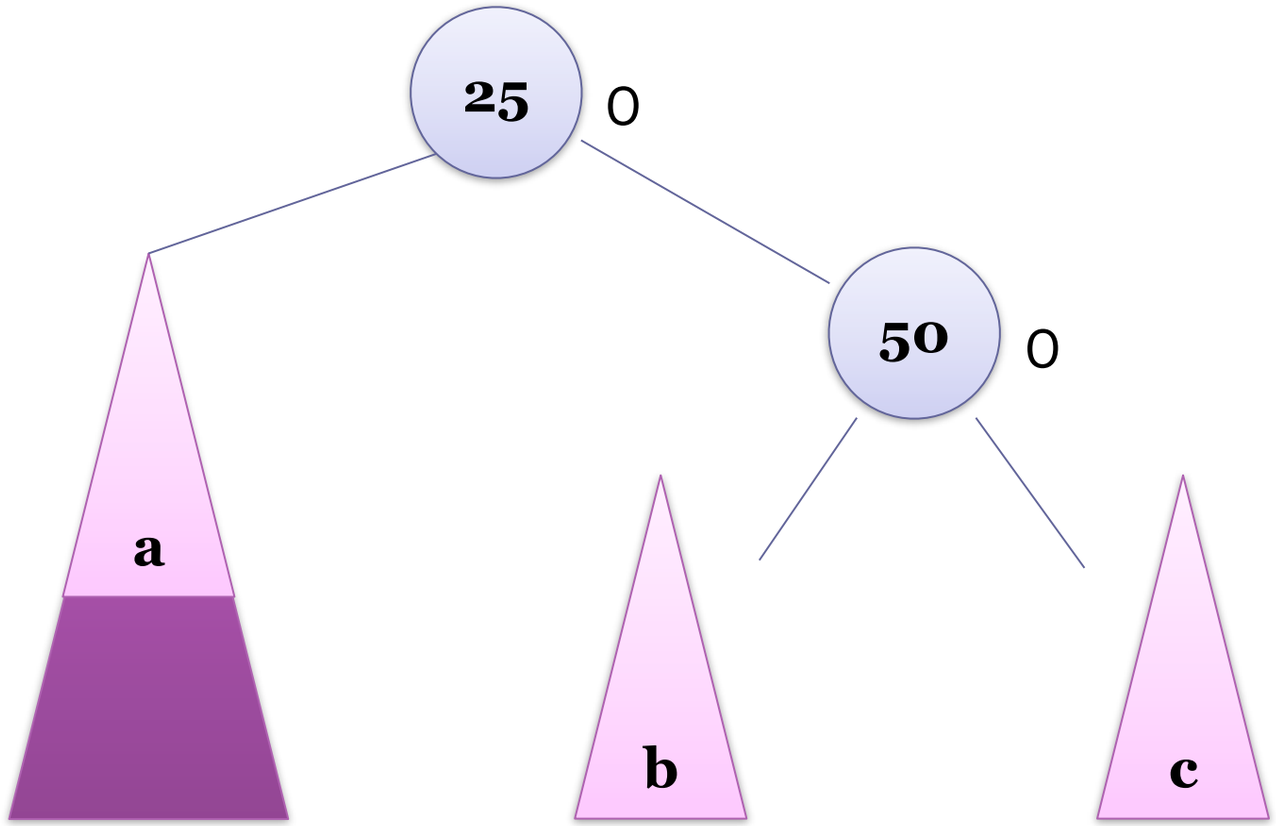


vänster-vänsterträd kan balanseras genom att rotera åt höger

Balansera ett vänster-vänsterträd

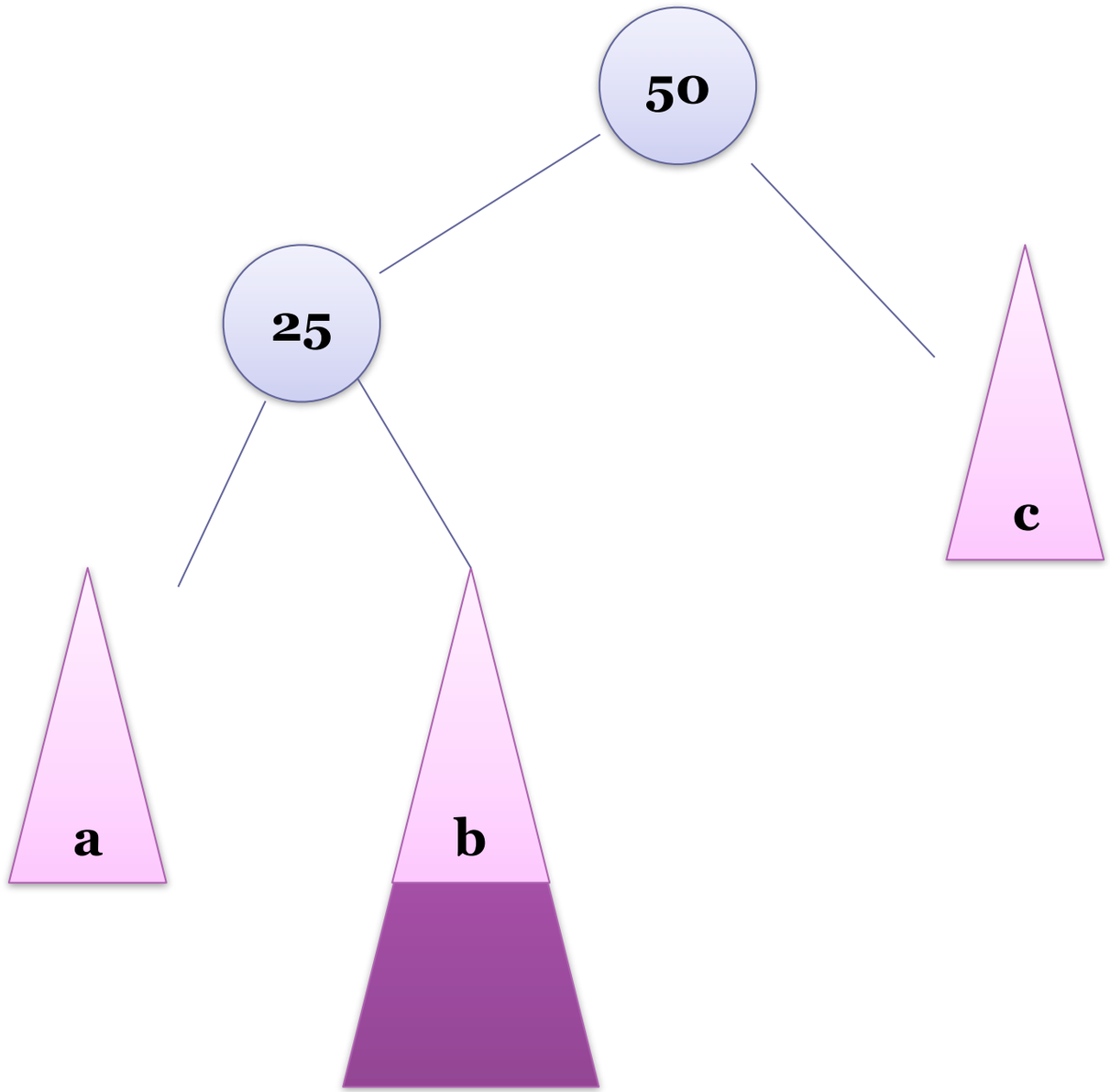


Balansera ett vänster-vänsterträd



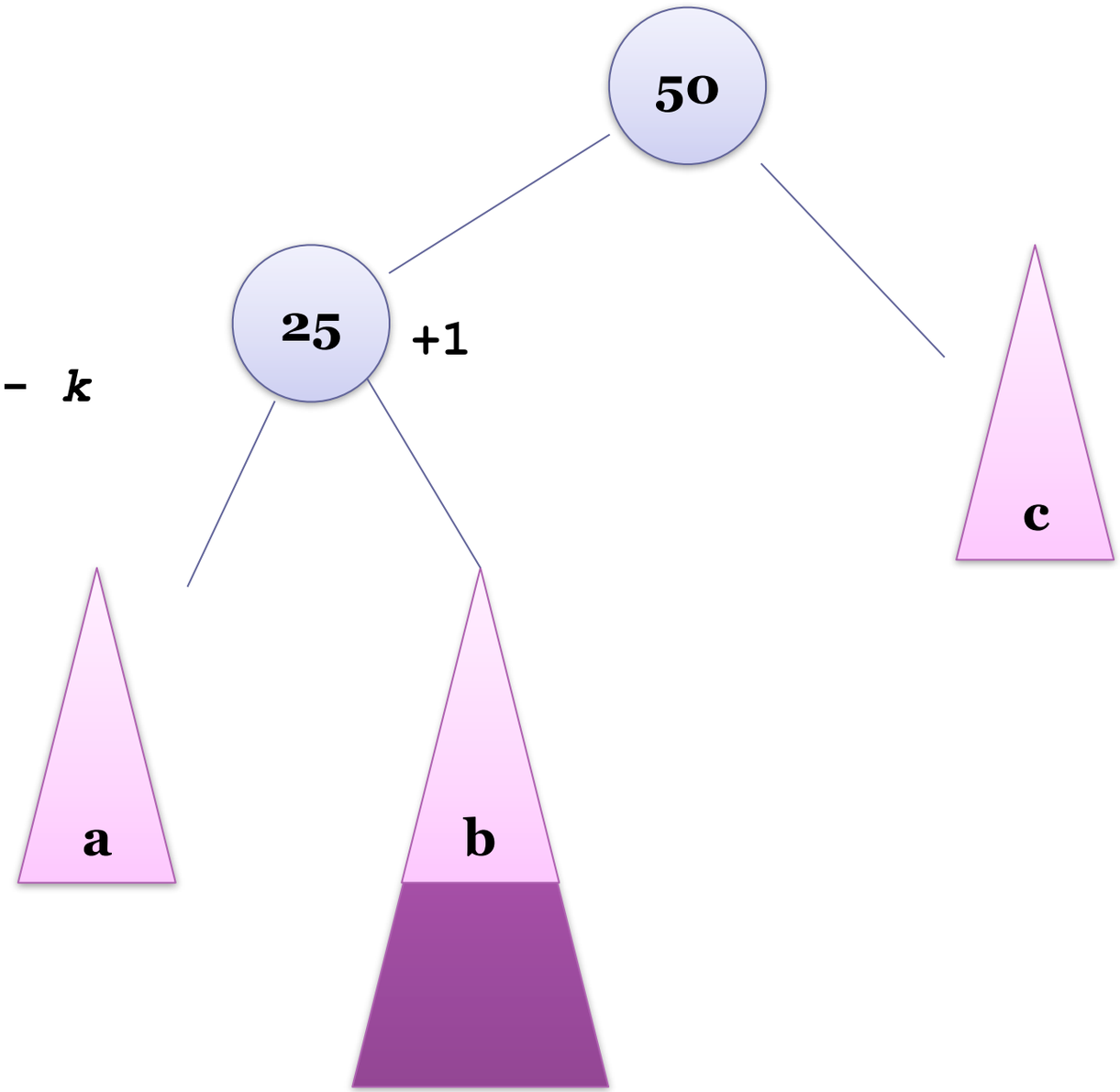
notera att den totala höjden *inte* har ändrats jämfört med innan insättningen!

Balansera ett vänster-högerträd

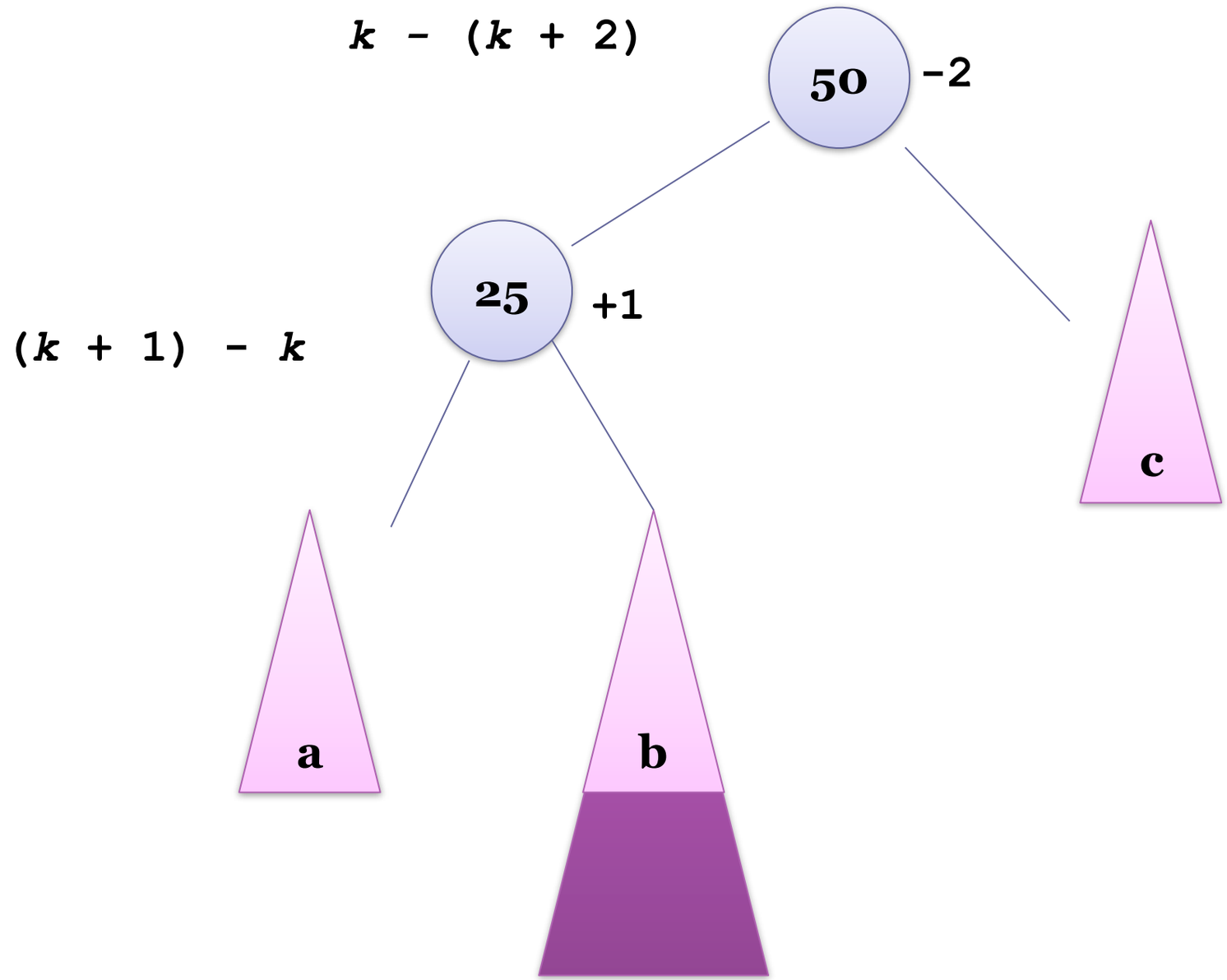


Balansera ett vänster-högerträd

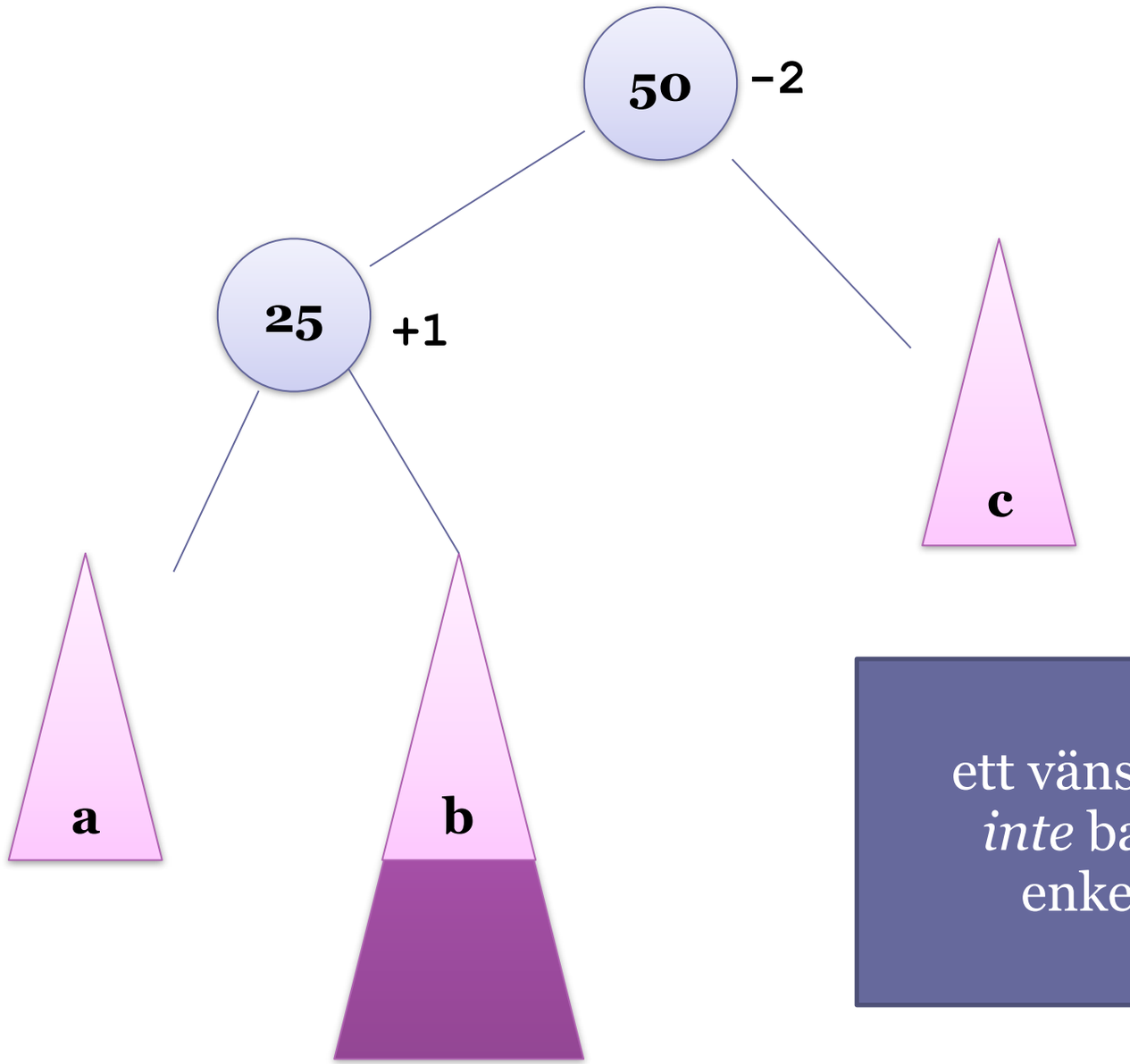
$(k + 1) - k$



Balansera ett vänster-högerträd

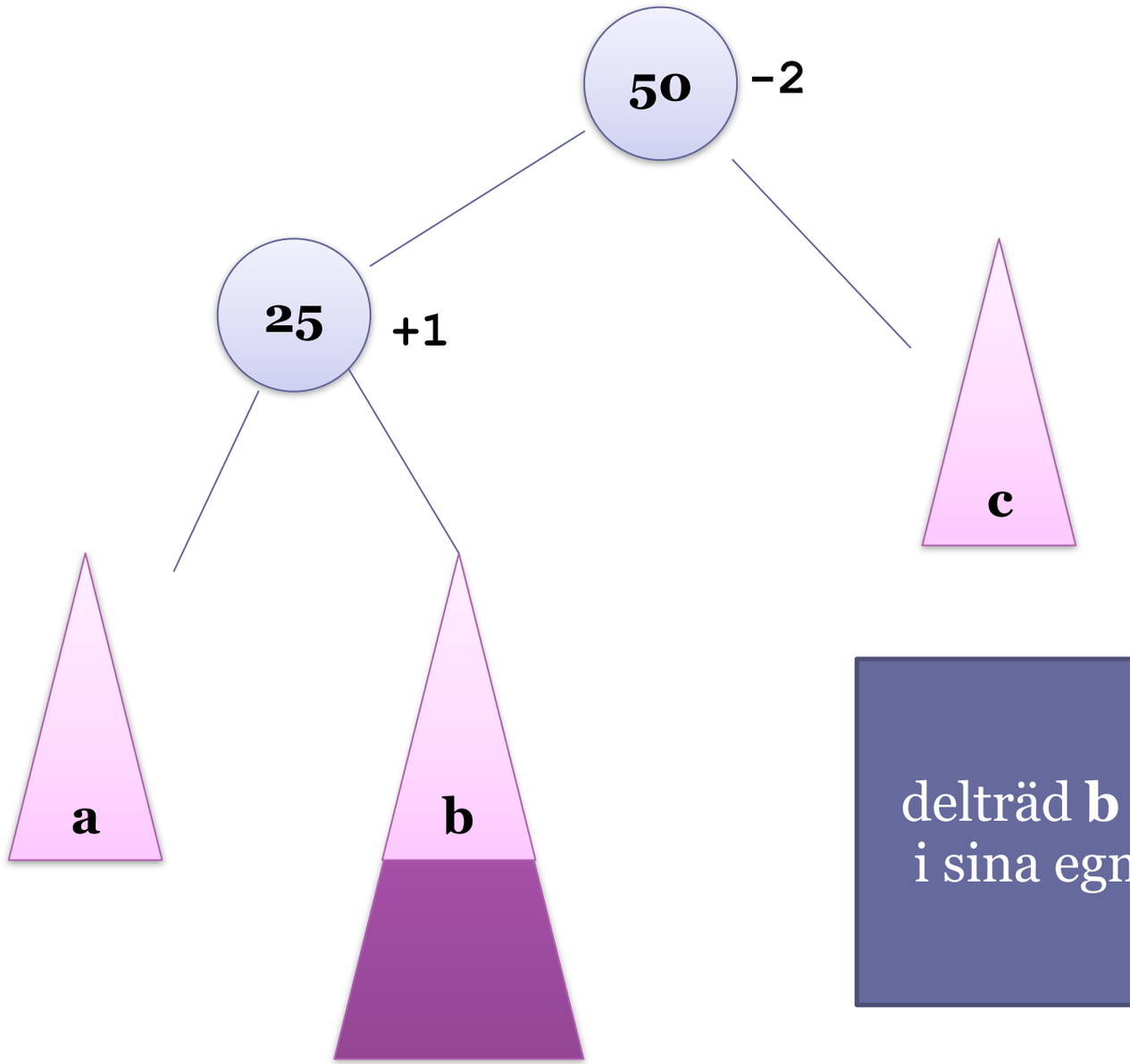


Balansera ett vänster-högerträd



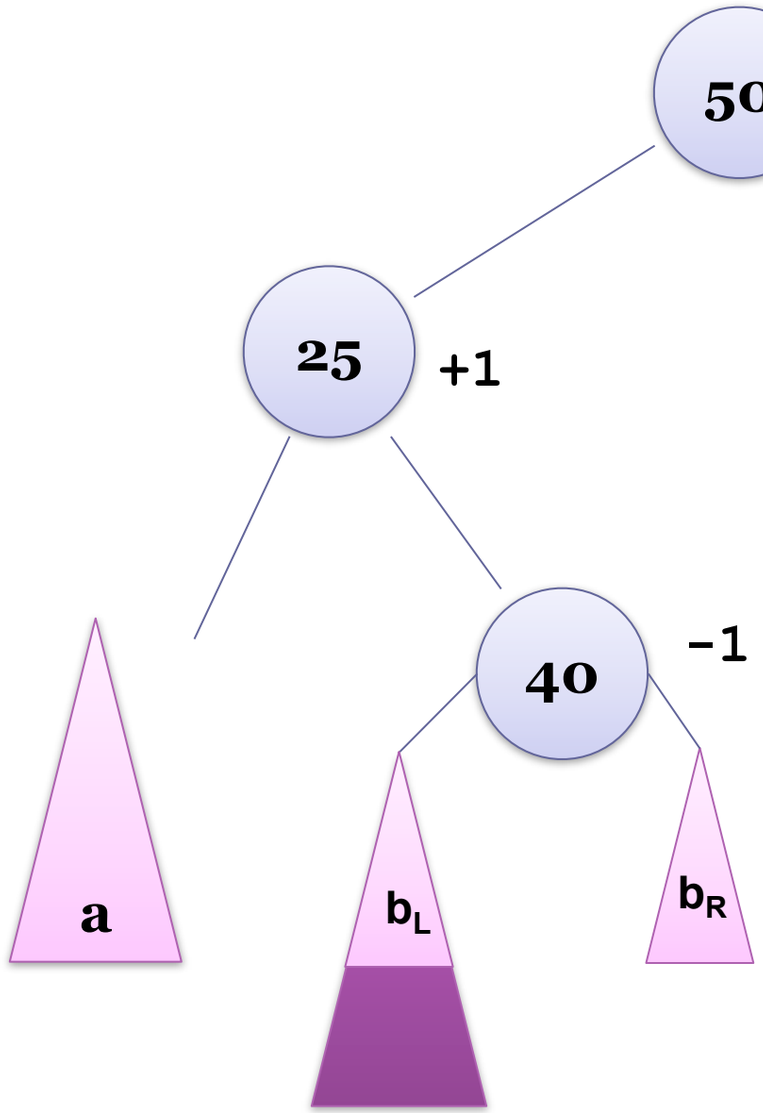
ett vänster-högerträd kan *inte* balanseras med en enkel högerrotation

Balansera ett vänster-högerträd



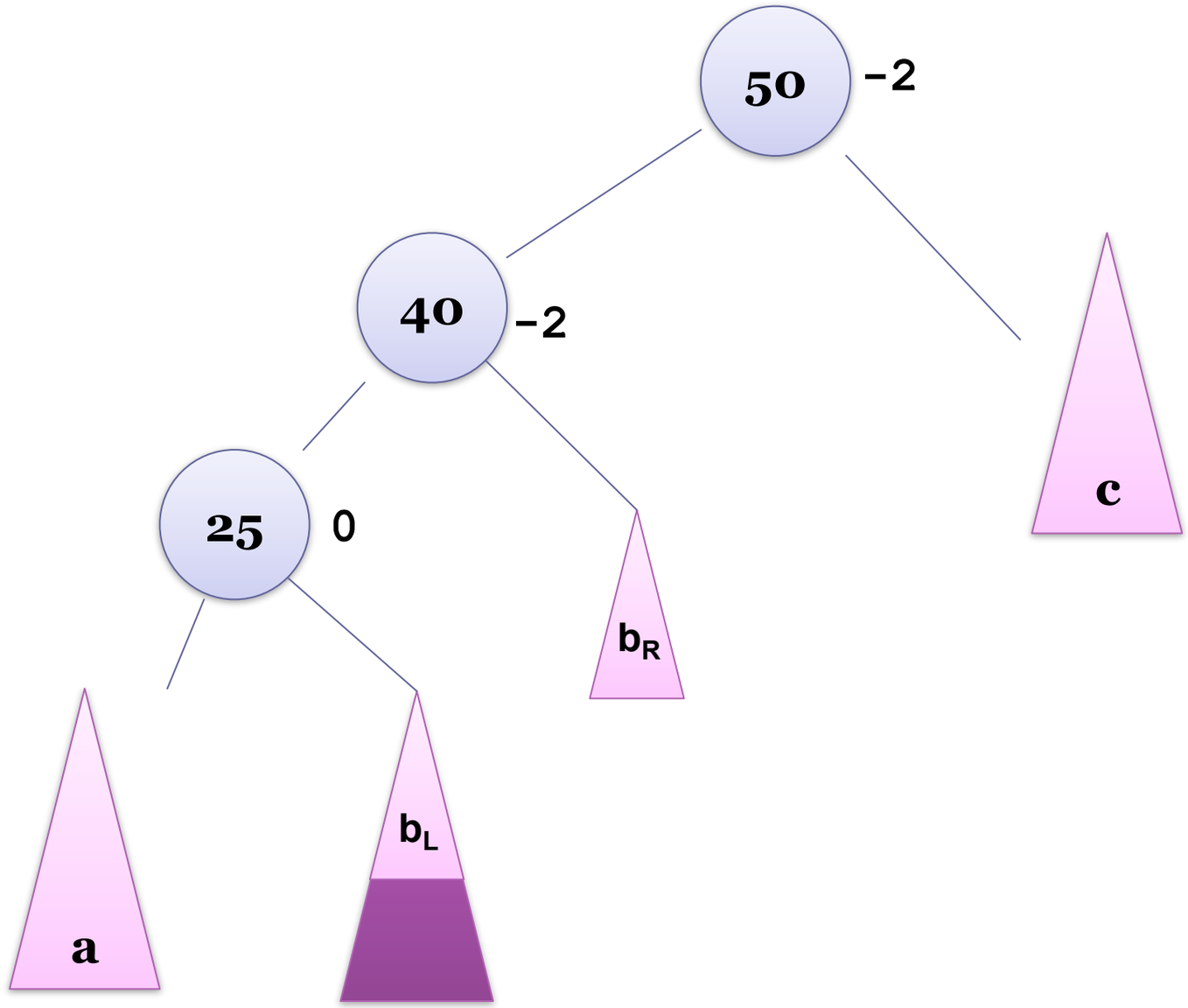
delträd **b** behöver expanderas i sina egna delträd $\mathbf{b_L}$ och $\mathbf{b_R}$

Balansera ett vänster-högerträd

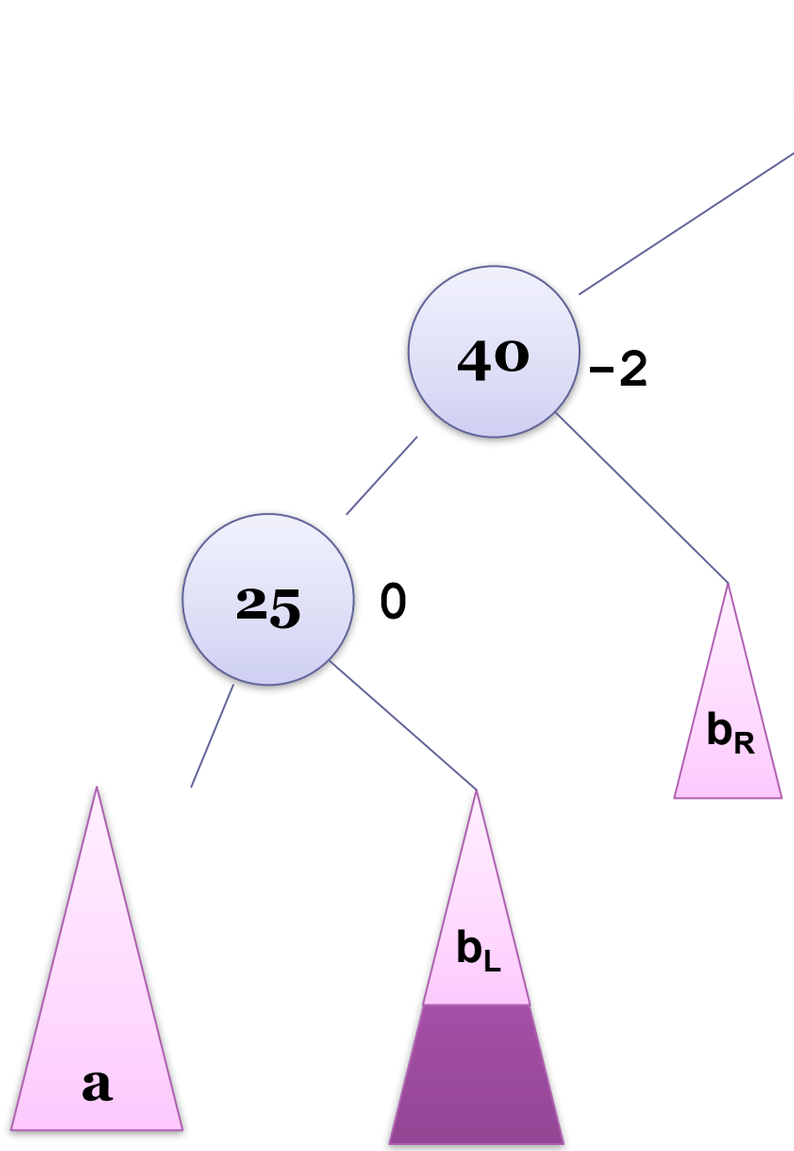


40 är vänster-tungt: vänster delträd kan nu roteras åt vänster

Balansera ett vänster-högerträd

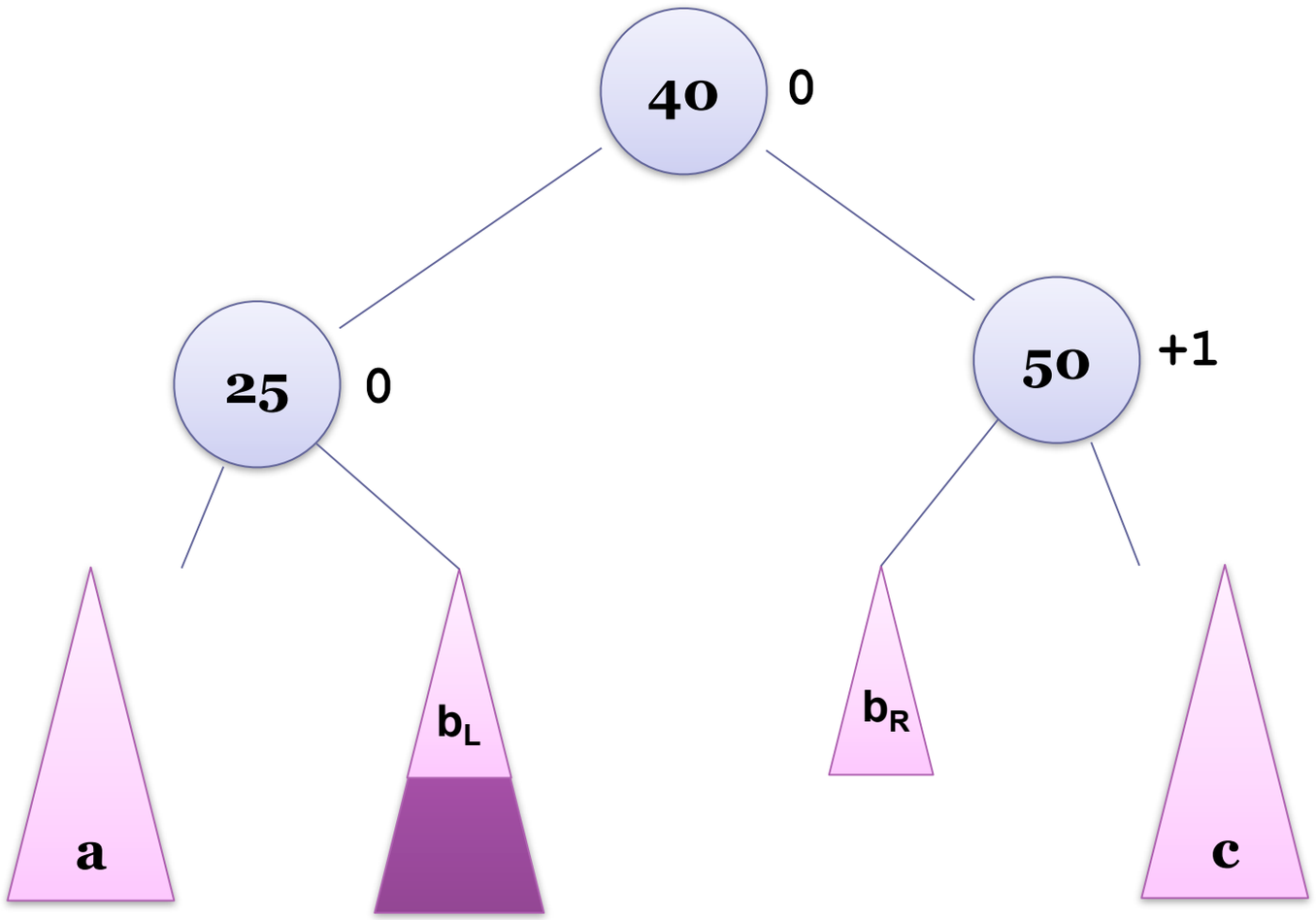


Balansera ett vänster-högerträd

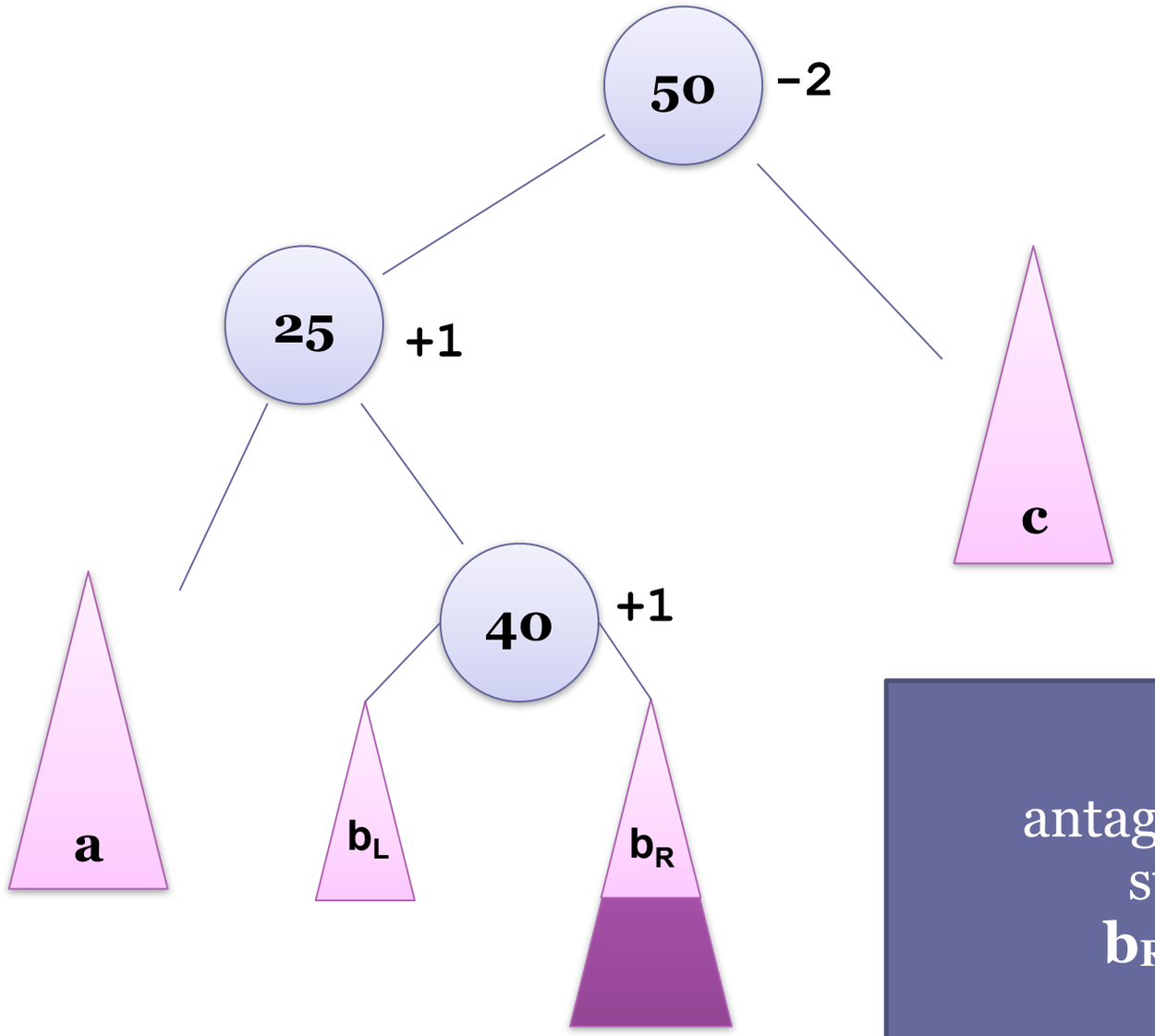


nu är det övergripande trädet vänster-vänster, så vi kan rotera åt höger för att balansera det

Balansera ett vänster-högerträd

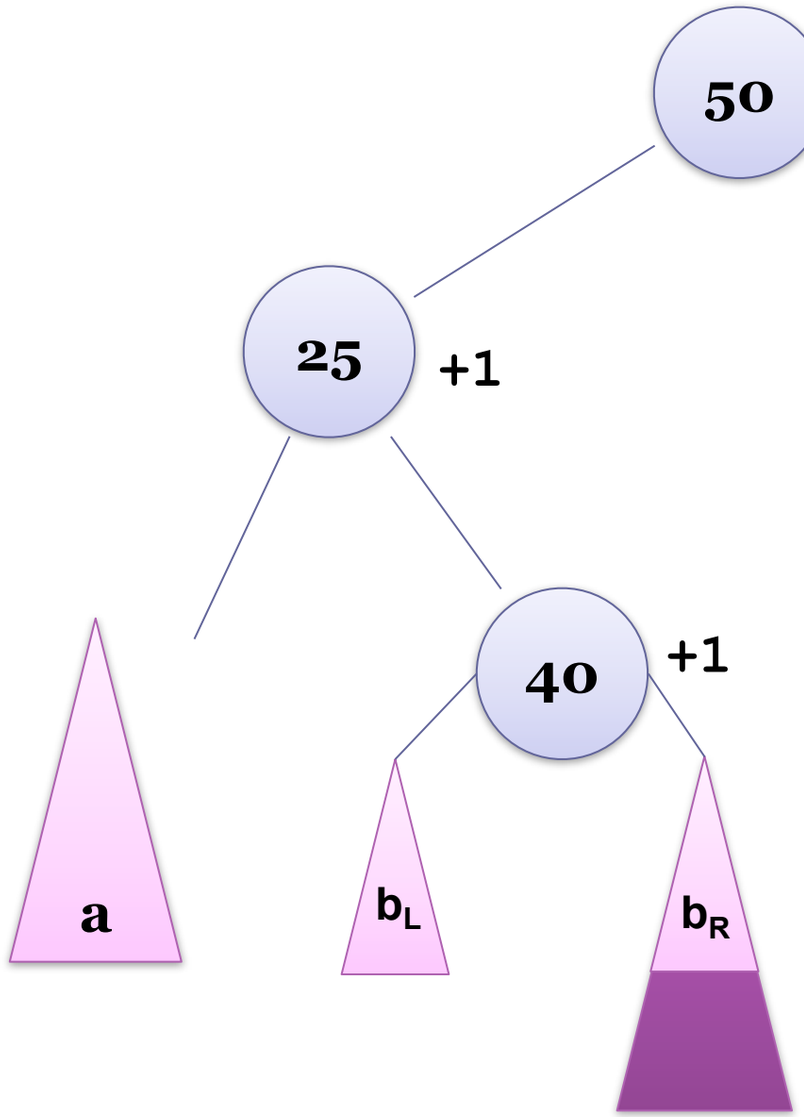


Balansera ett vänster-högerträd



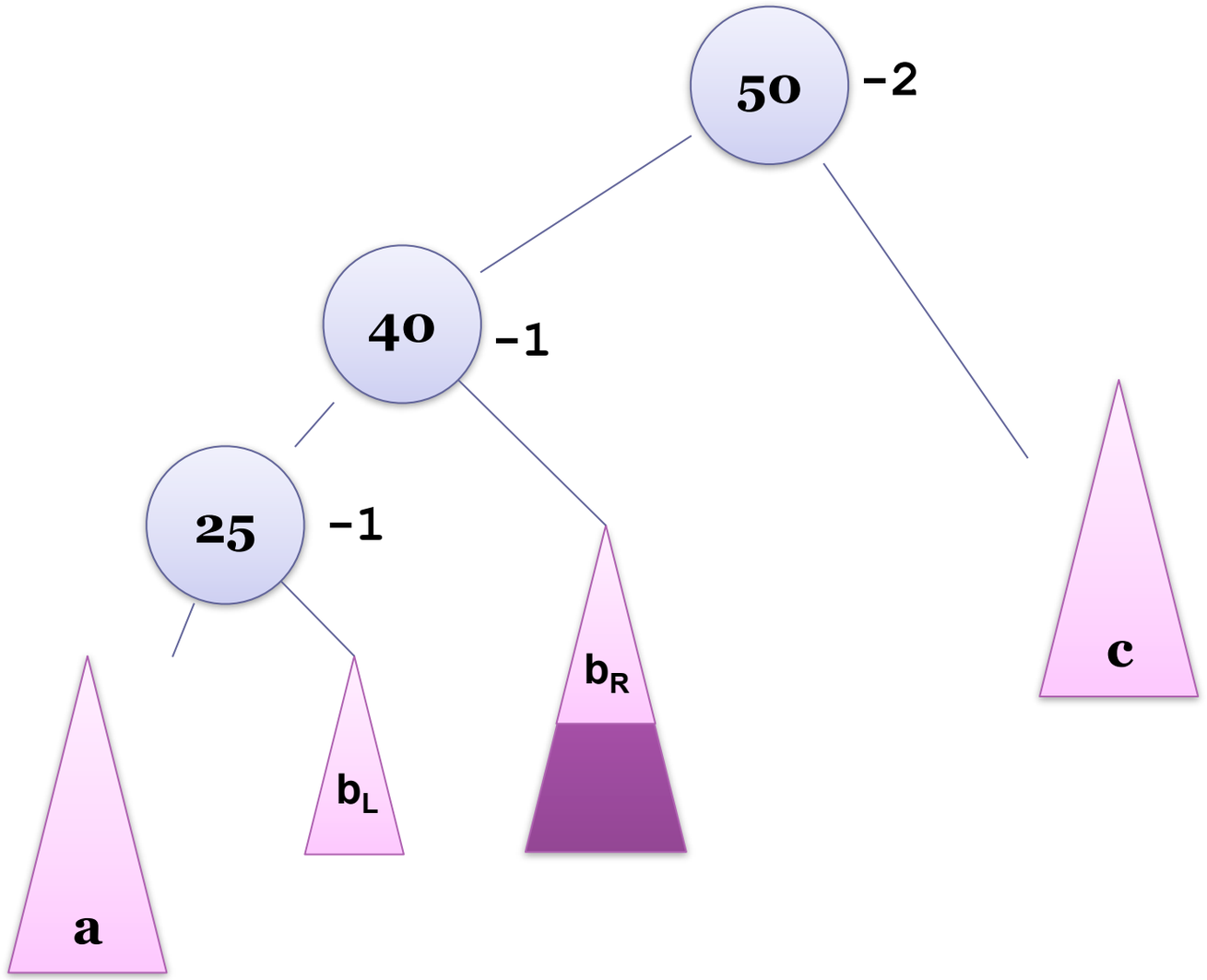
antag nu att elementet stoppades in i $\mathbf{b_R}$ istället för $\mathbf{b_L}$

Balansera ett vänster-högerträd

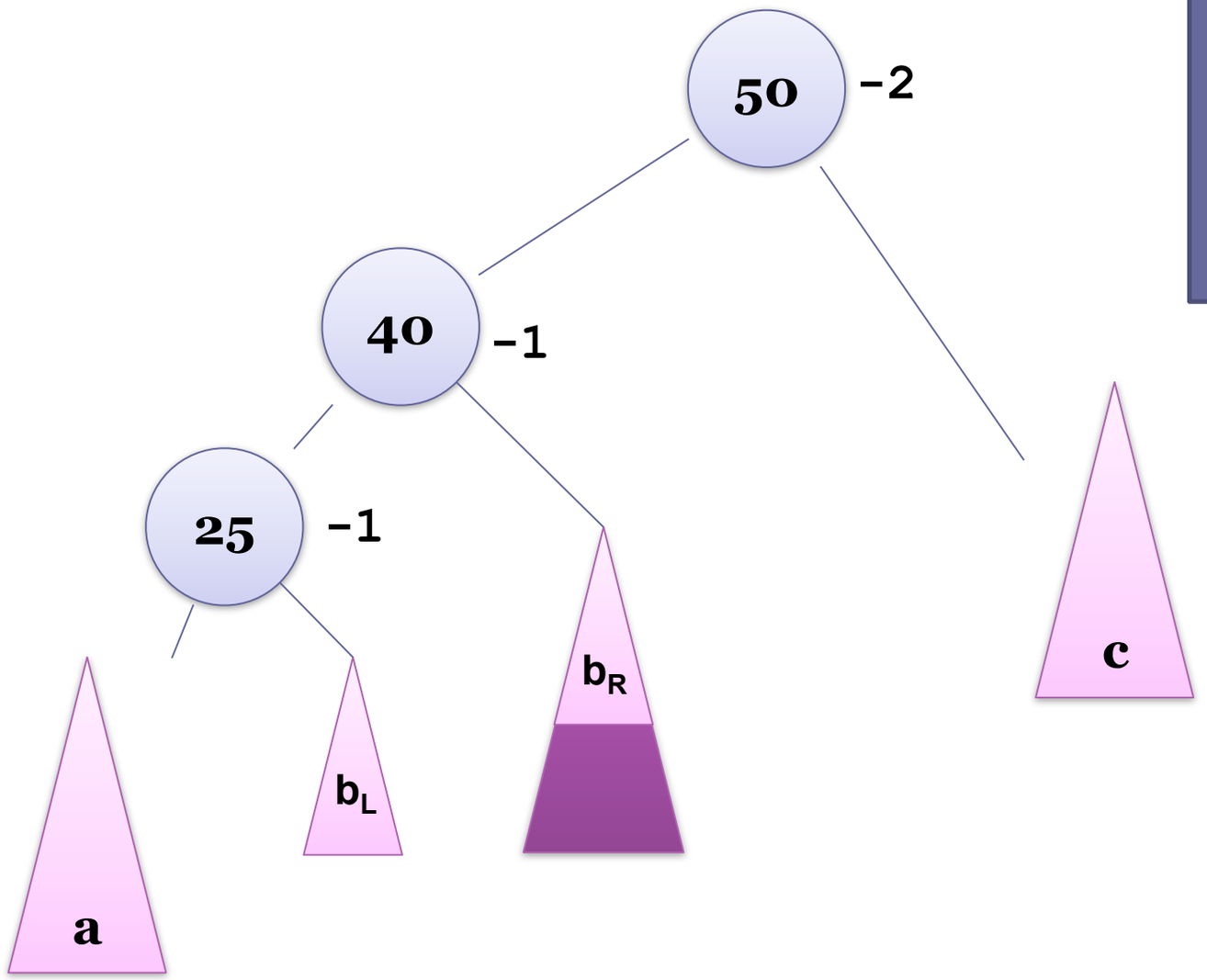


1. rotera vänster
delträd åt vänster

Balansera ett vänster-högerträd

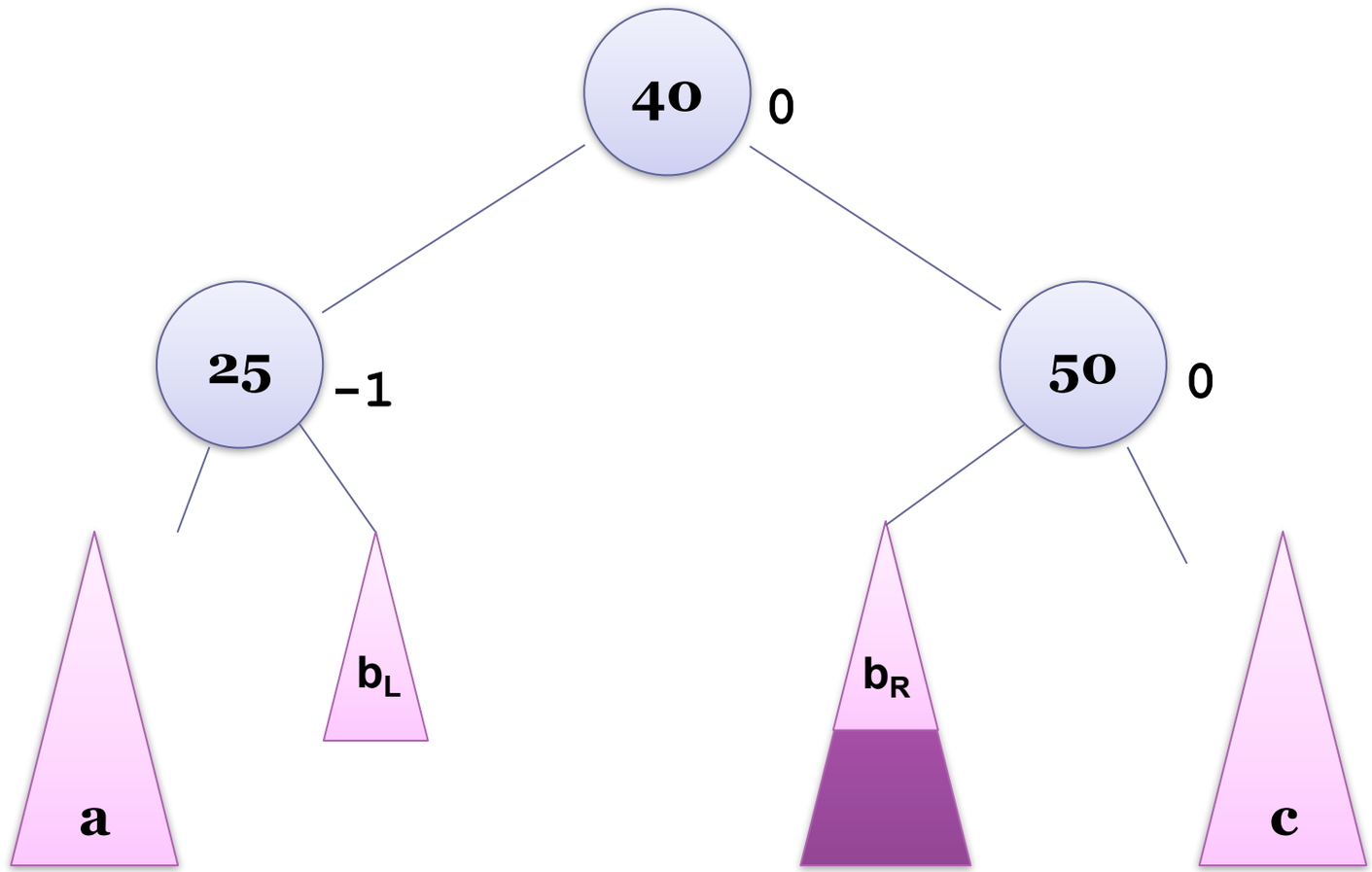


Balansera ett vänster-högerträd



2. rotera trädet åt höger

Balansera ett vänster-högerträd



Fyra sorters obalanserade träd

Vänster-vänster (föräldern < −1, vänster barn ≤ 0)

- rotera åt höger runt föräldern

Vänster-höger (föräldern < −1, vänster barn > 0)

- rotera åt vänster runt barnet
- rotera åt höger runt föräldern

Höger-höger (föräldern > 1, höger barn ≥ 0)

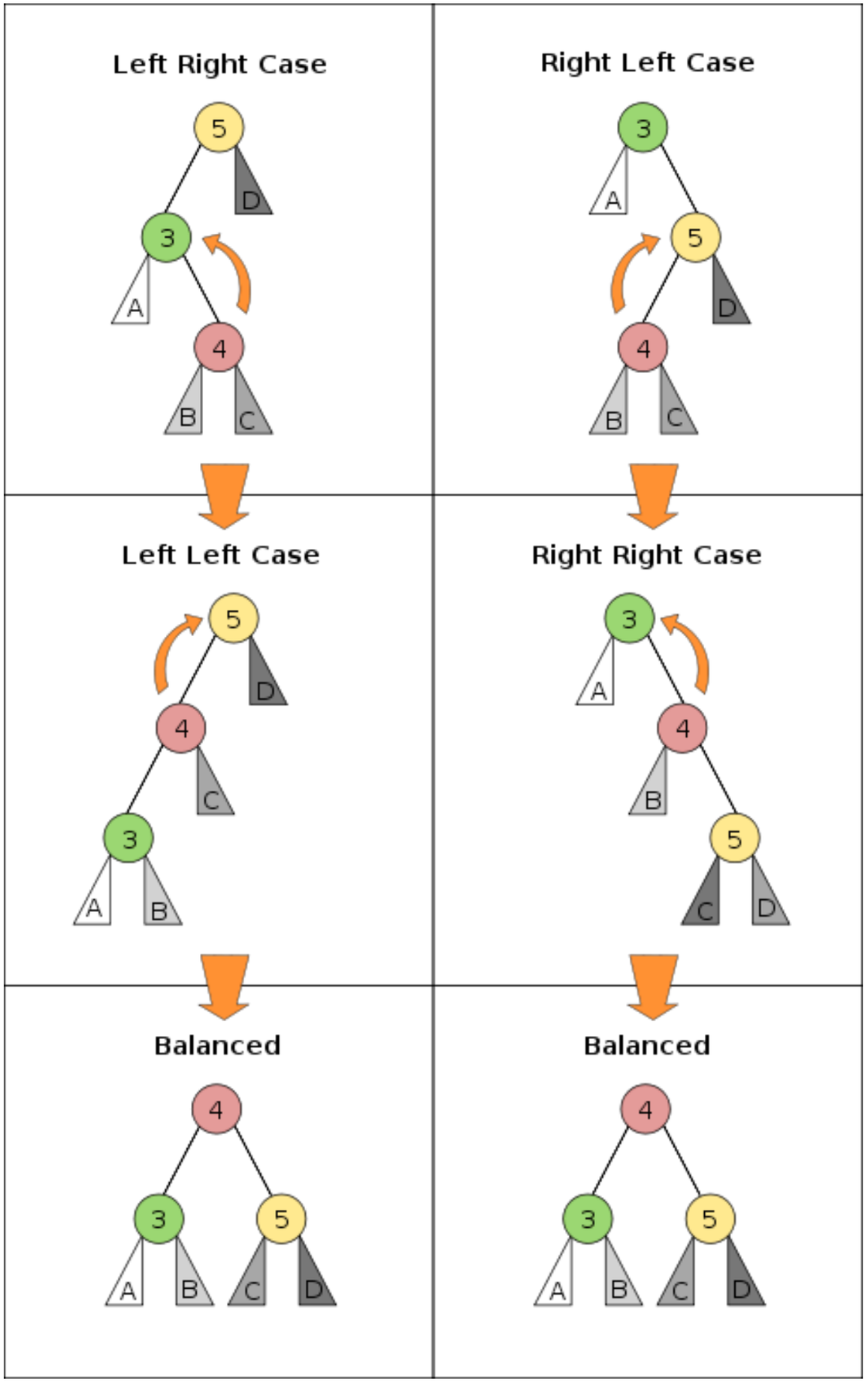
- rotera åt vänster runt föräldern

Höger-vänster (föräldern > 1, höger barn < 0)

- rotera åt höger runt barnet
- rotera åt vänster runt föräldern

Balansering av de fyra olika fallen

(bilden är från Wikipedia)

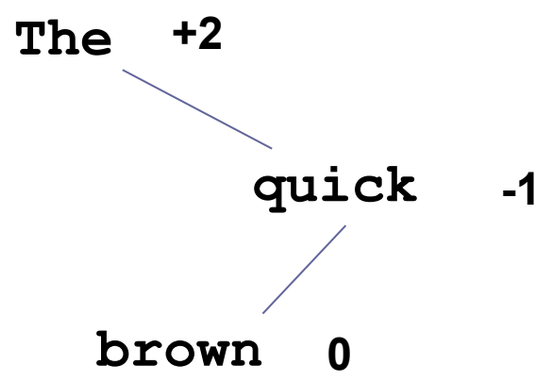


Ett större AVL-exempel

Nu ska vi bygga ett AVL-träd för orden i

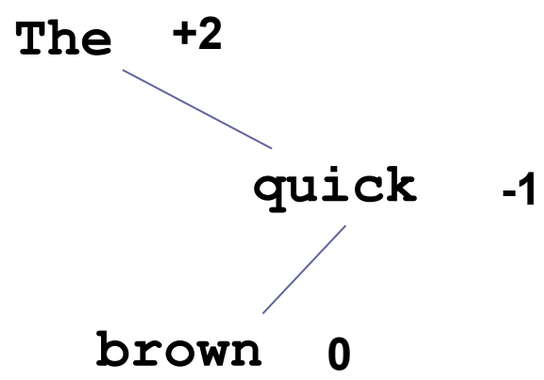
”The quick brown fox jumps over the lazy dog”

The quick brown…



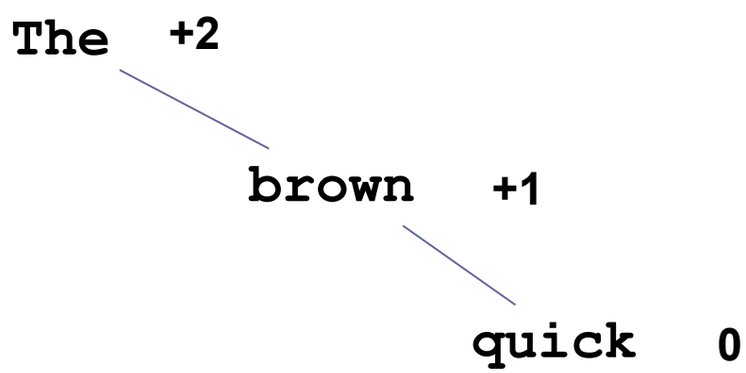
The overall tree is right-heavy (Right-Left)
parent balance = +2
right child balance = -1

The quick brown…



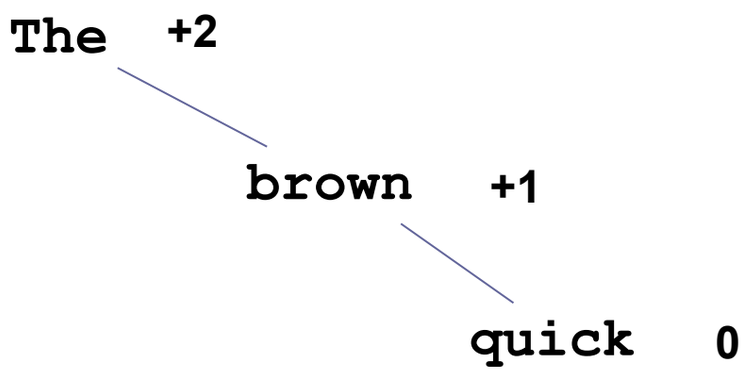
1. **Rotate right around the child**

The quick brown…



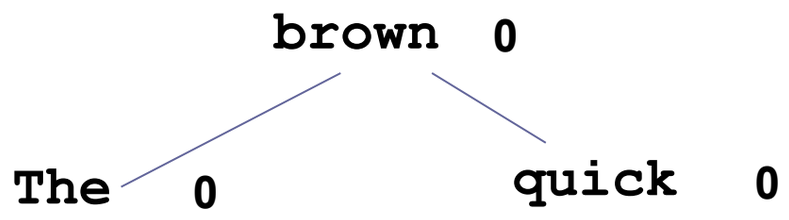
1. **Rotate right around the child**

The quick brown…



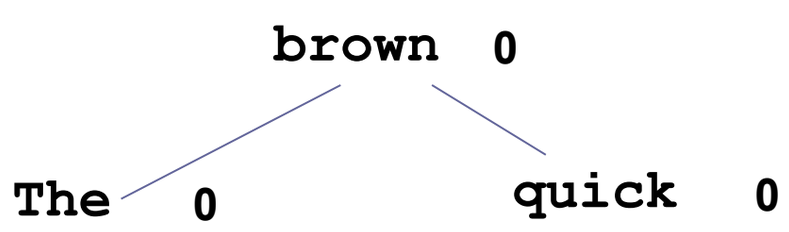
1. Rotate right around the child
2. Rotate left around the parent

The quick brown…



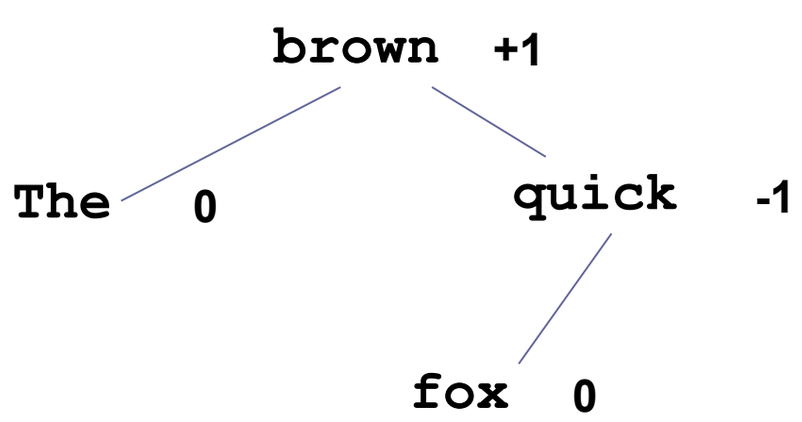
1. Rotate right around the child
2. Rotate left around the parent

The quick brown fox…



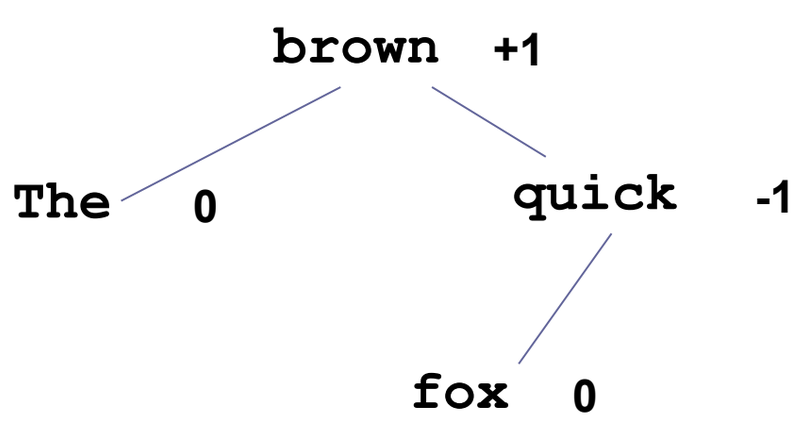
Insert *fox*

The quick brown fox…



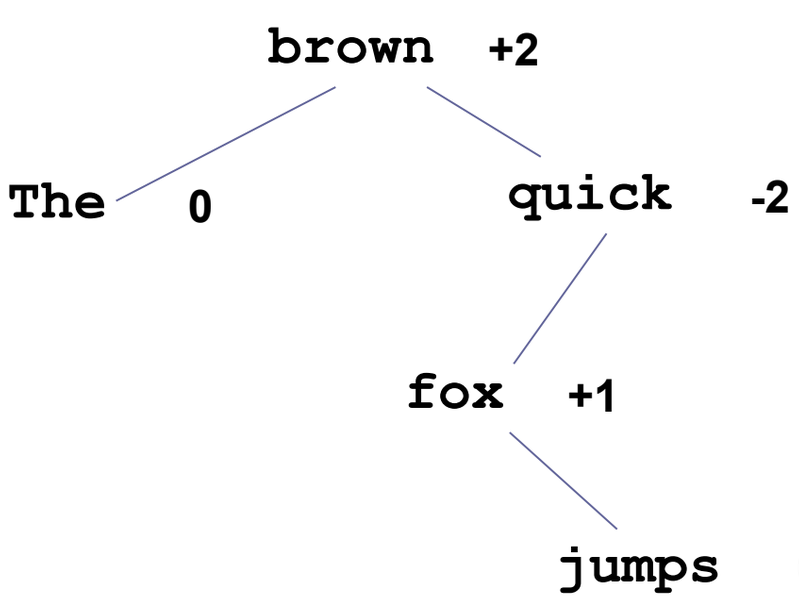
Insert *fox*

The quick brown fox jumps…



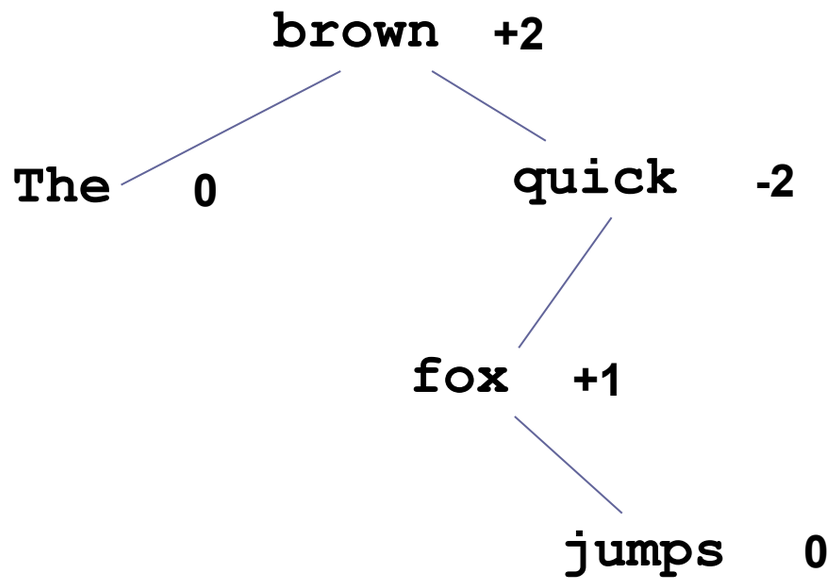
Insert *jumps*

The quick brown fox jumps…



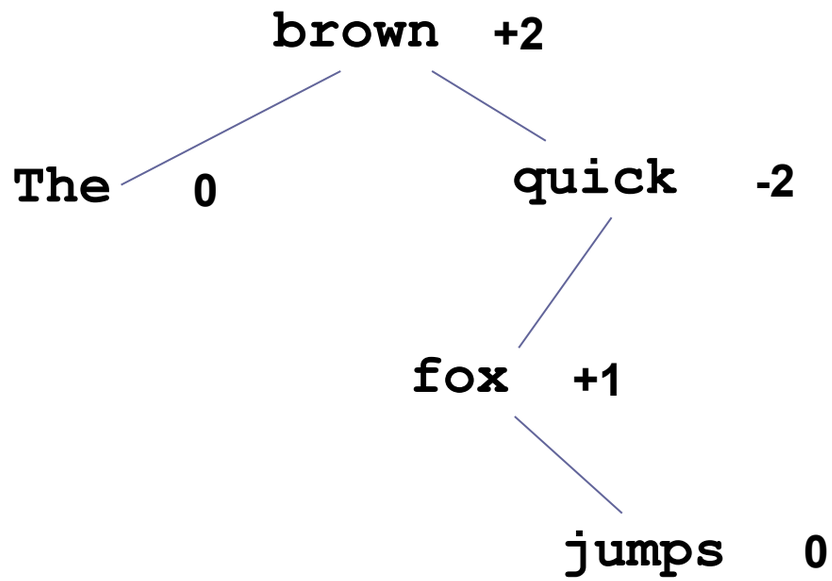
Insert *jumps*

The quick brown fox jumps…



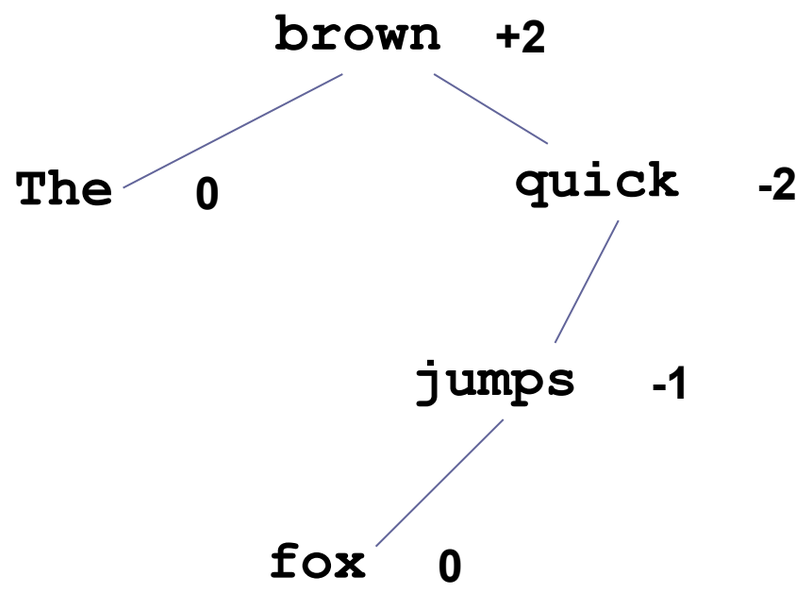
The tree is now left-heavy about *quick* (Left-Right case)

The quick brown fox jumps…



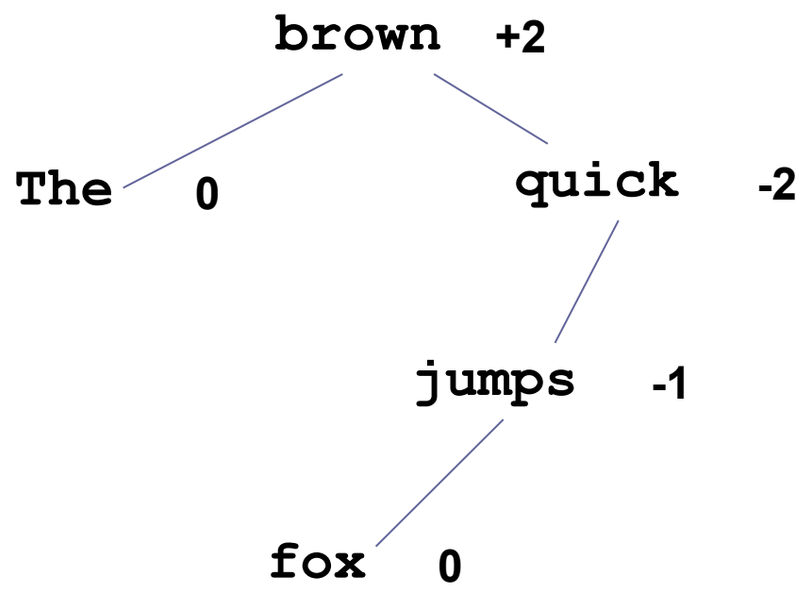
1. **Rotate left around the child**

The quick brown fox jumps…



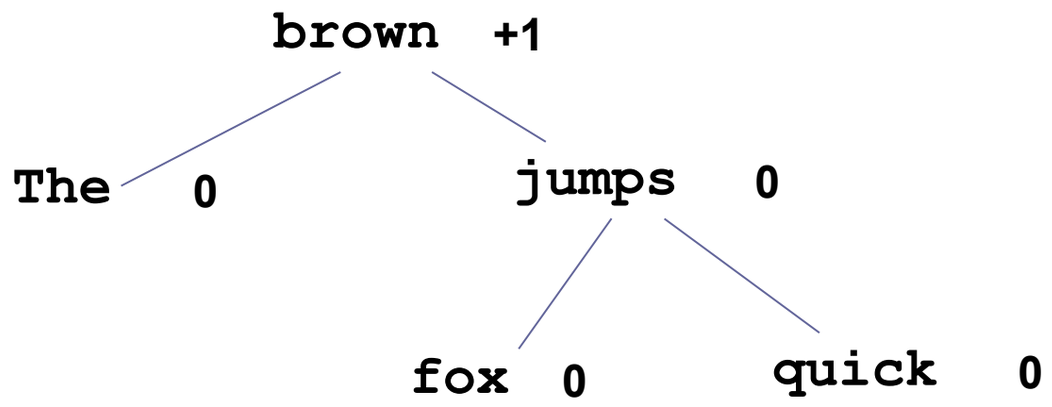
1. **Rotate left around the child**

The quick brown fox jumps…



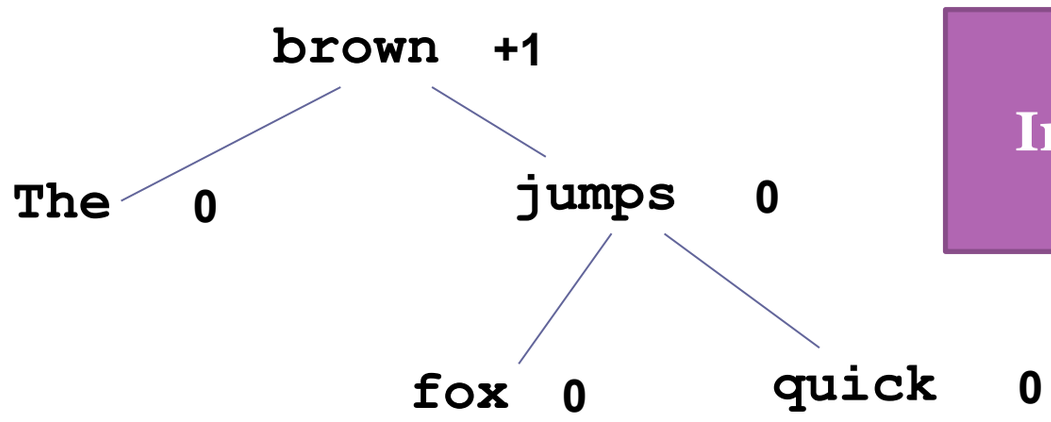
1. Rotate left around the child
2. Rotate right around the parent

The quick brown fox jumps…



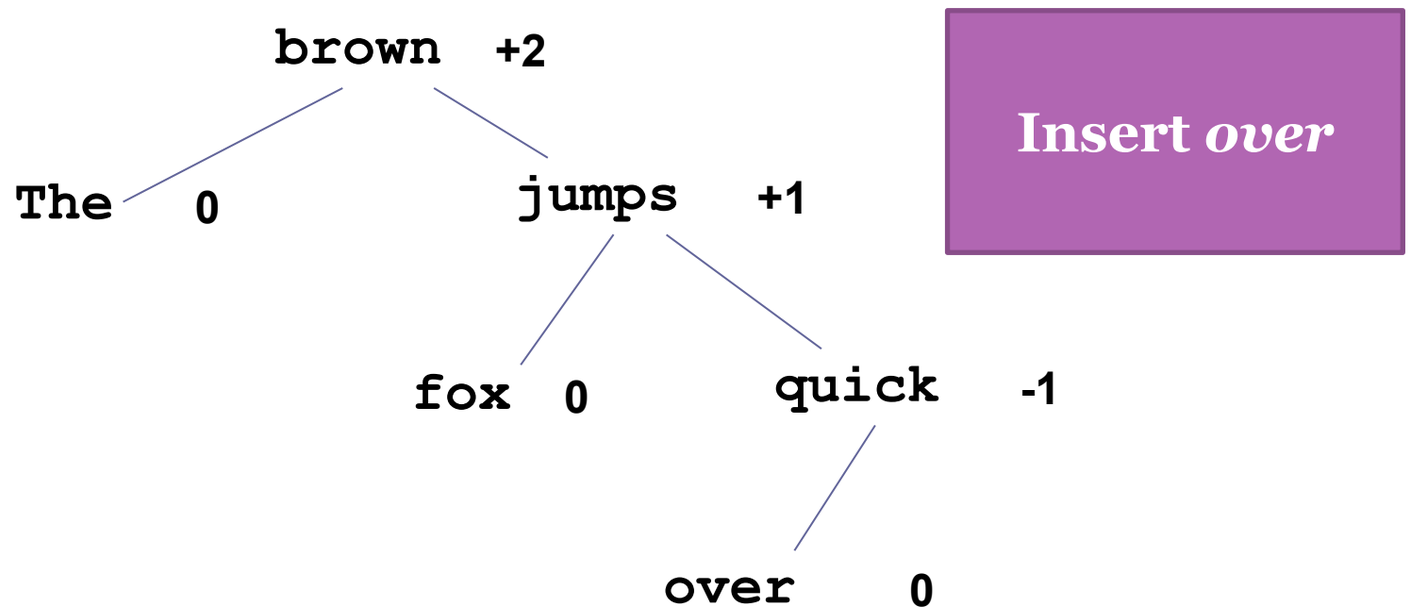
1. Rotate left around the child
2. Rotate right around the parent

The quick brown fox jumps over…

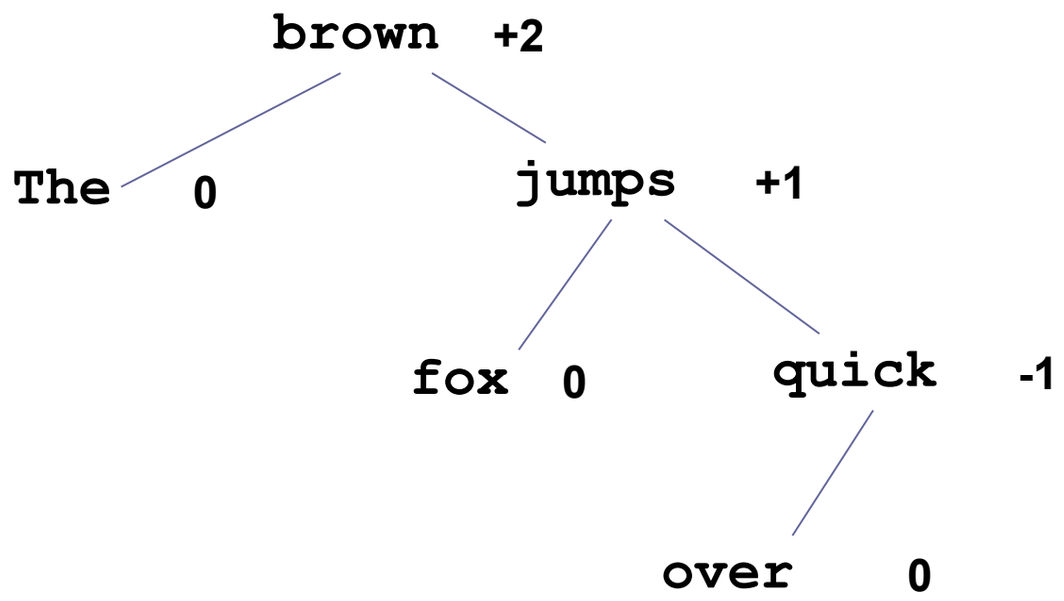


Insert *over*

The quick brown fox jumps over…

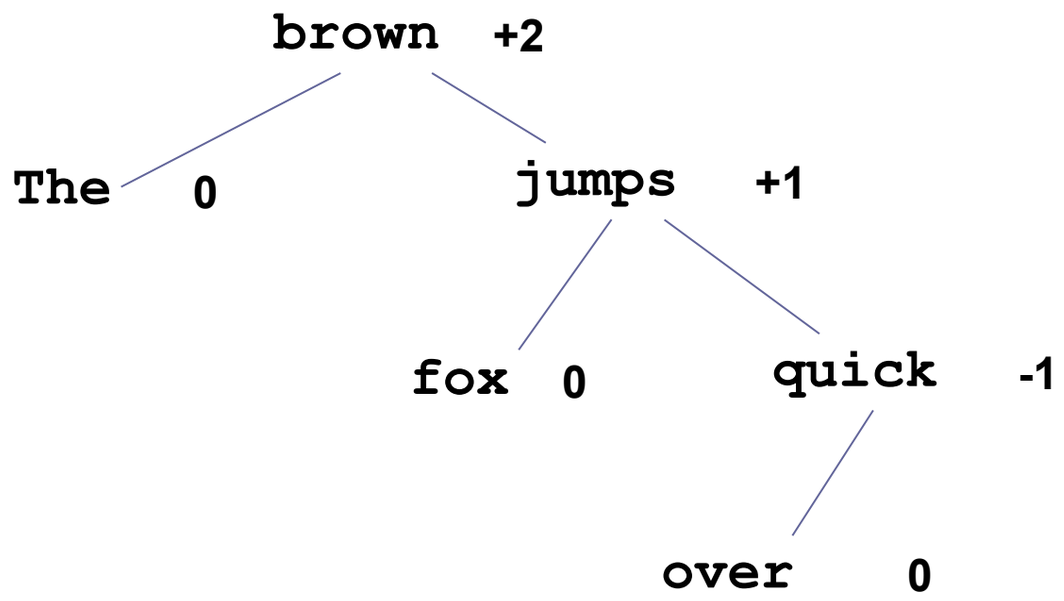


The quick brown fox jumps over…



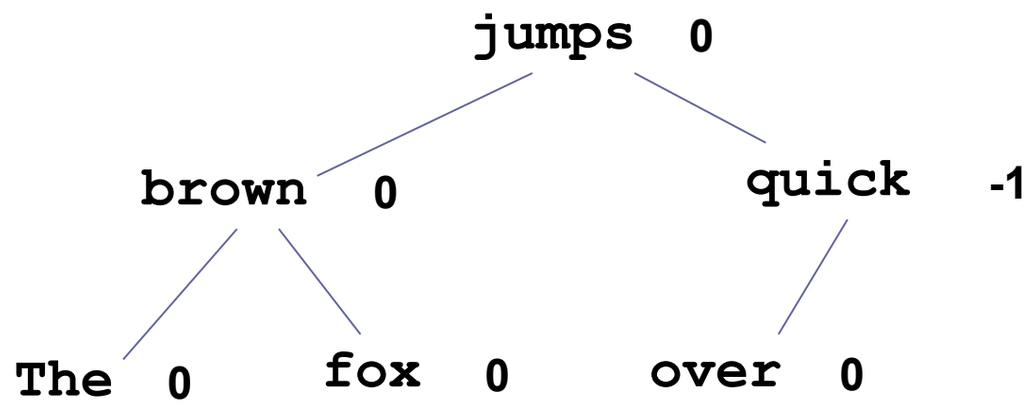
We now have a Right-Right imbalance

The quick brown fox jumps over…



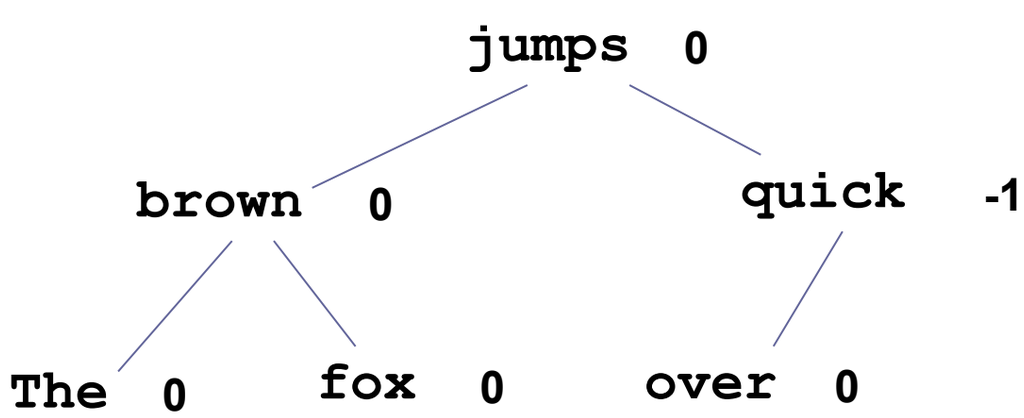
1. Rotate left around the parent

The quick brown fox jumps over…



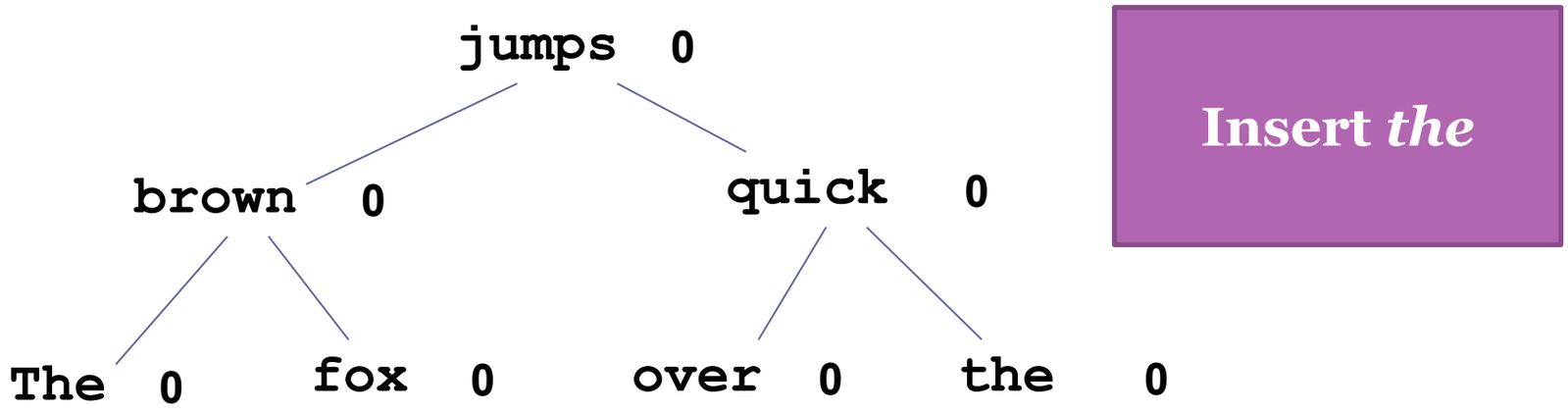
1. Rotate left around the parent

The quick brown fox jumps over the…

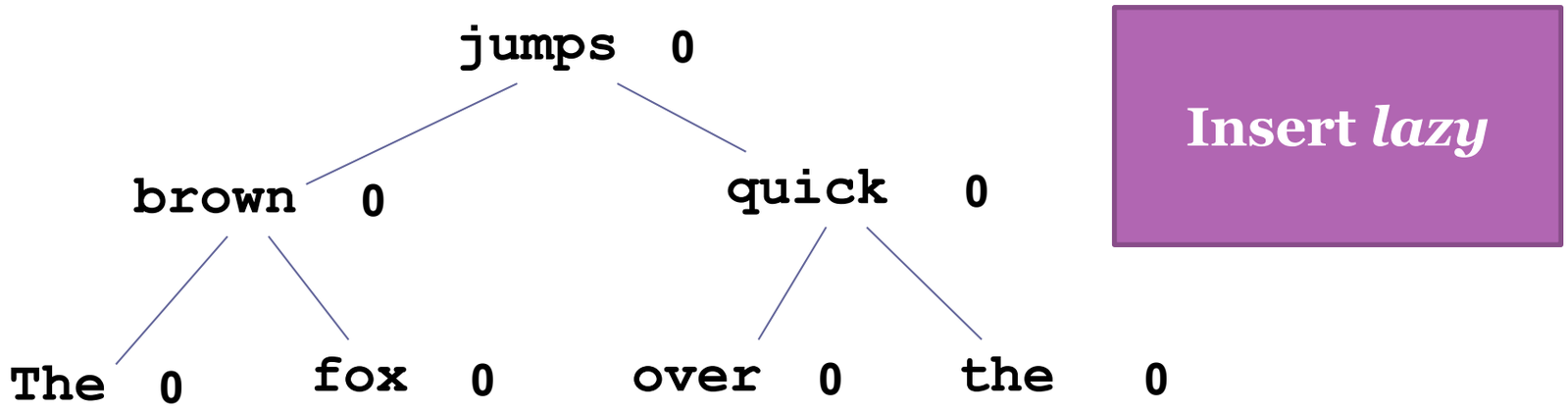


Insert *the*

The quick brown fox jumps over the…

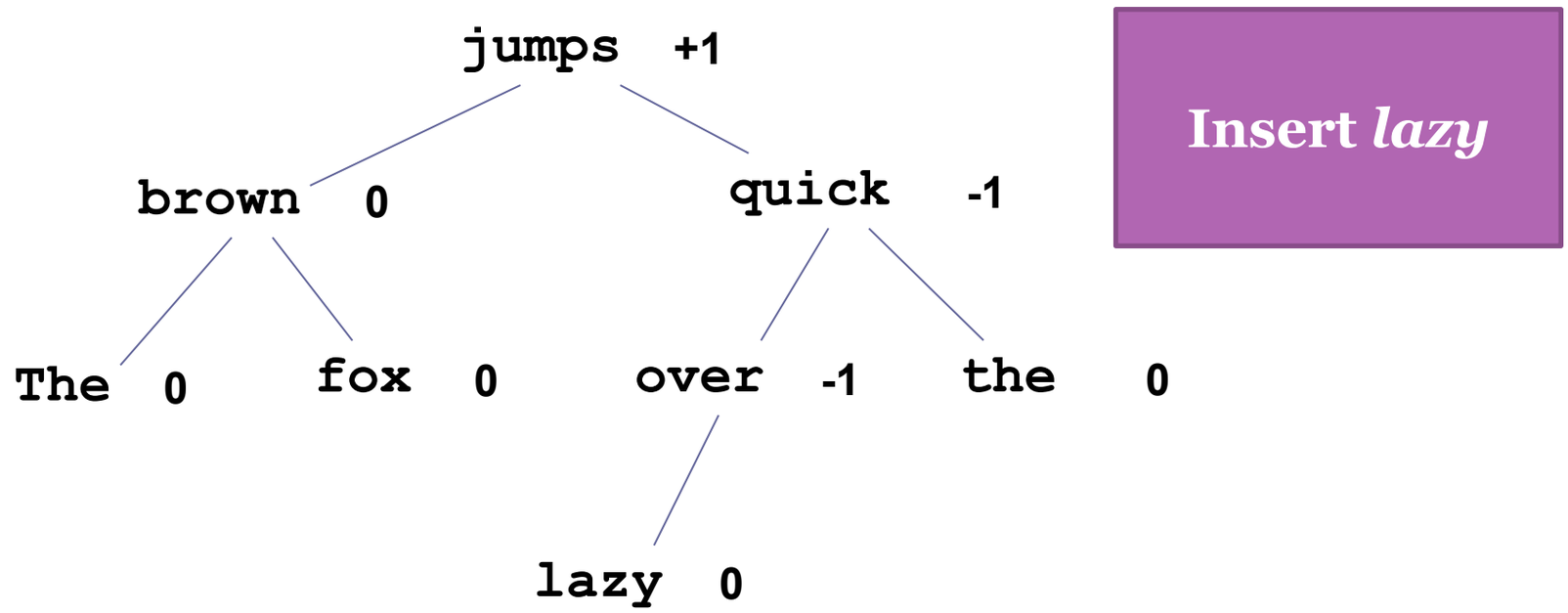


quick brown fox jumps over the lazy…



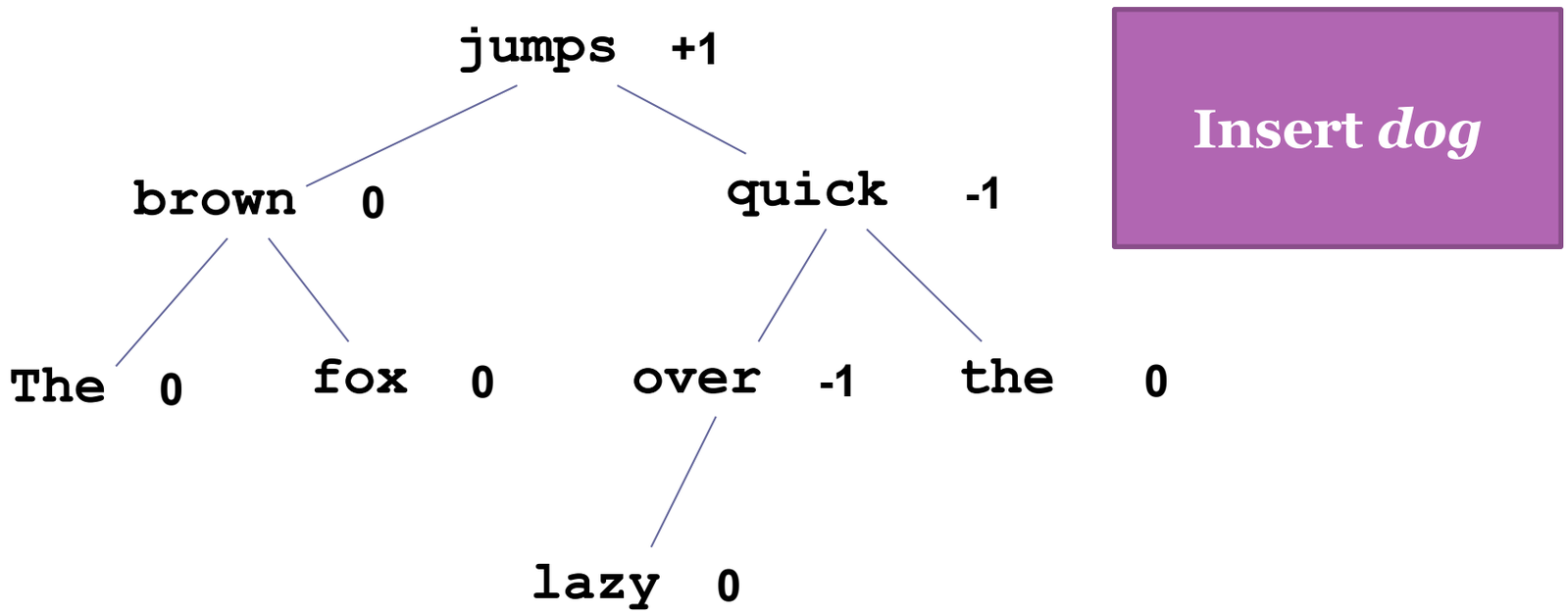
Insert *lazy*

quick brown fox jumps over the lazy…

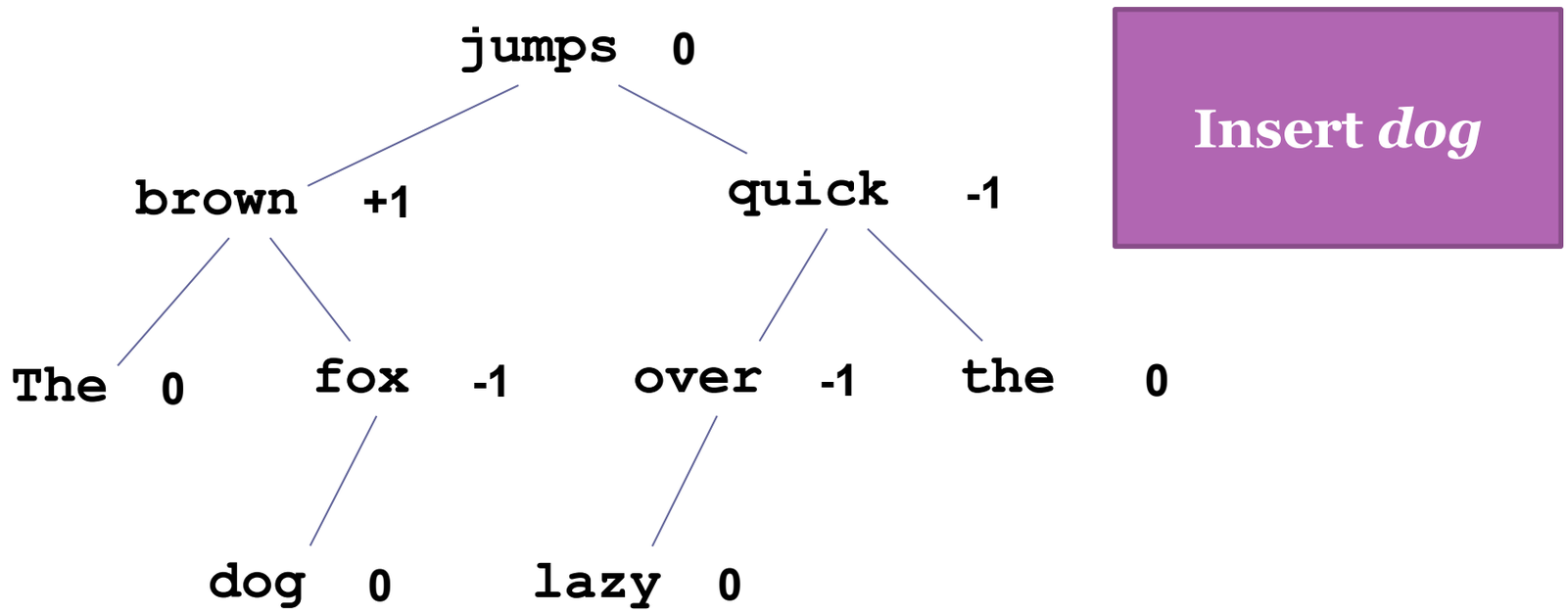


Insert *lazy*

uick brown fox jumps over the lazy dog

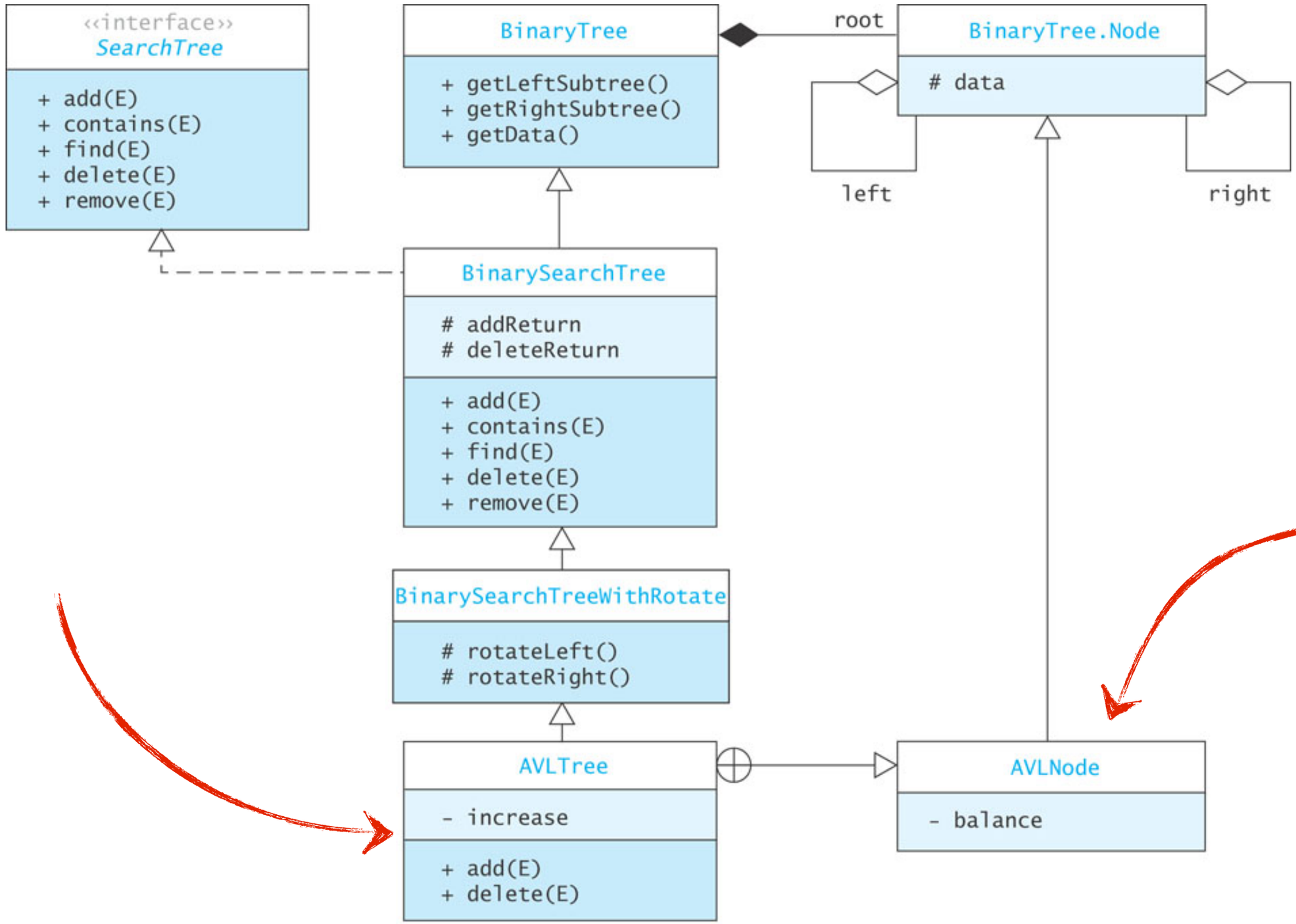


ick brown fox jumps over the lazy dog!



Insert *dog*

Att implementera ett AVL-träd



Insättning i ett AVL-träd

Enklaste sättet att hålla trädet balanserat är att aldrig låta det bli obalanserat

- om någon nod blir kritiskt obalanserad, balansera om den direkt
- identifiera kritiska noder genom att kontrollera nodbalansen när du returnerar efter att ha lagt in det nya elementet

Insättning i ett AVL-träd

Algoritm för insättning i ett AVL-träd

1. if the root is null:
2. 	Create a new tree with the item at the root and return true
3. else if the item is equal to root.data:
4. 	The item is already in the tree; return false
5. else if the item is less than root.data:
6. 	Recursively insert the item in the left subtree
7. 	if the height of the left subtree has increased (increase is true):
8. 		Decrement balance
9. 		if balance is zero, reset increase to false
10. 		if balance is less than -1:
11. 			Reset increase to false
12. 			Perform a rebalanceLeft
13. else if the item is greater than root.data:
14. 	The processing is symmetric to Steps 4 through 10.
15. 	Note that balance is incremented if increase is true, not decremented

rebalanceLeft: första försöket

Initial algoritm för rebalanceLeft

1. if the left subtree has positive balance (Left-Right case):
2. Rotate left around left subtree root
3. Rotate right

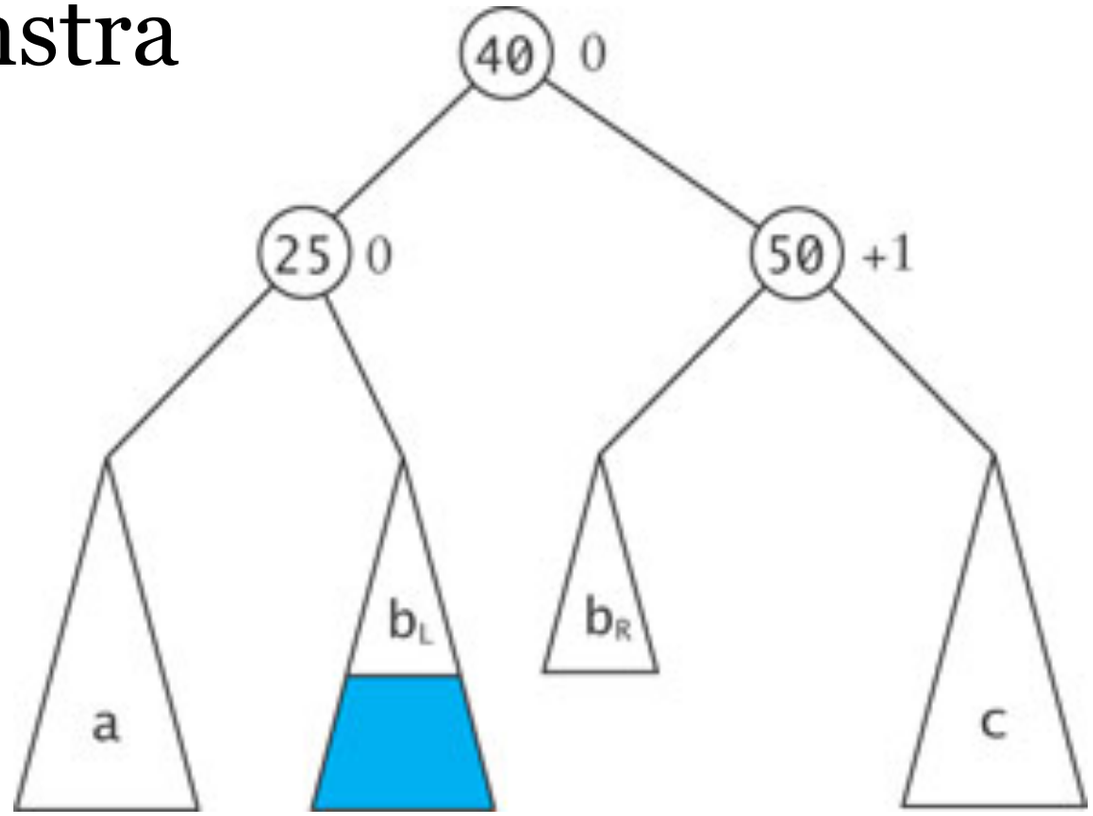
Denna algoritm är inte fullständig, eftersom nodbalanserna inte justerades.

- för ett vänster-vänsterträd så blir den slutliga balansen 0 på rotnoden och dess högra barn
- vänster-högerträd är mer komplicerade

Rotationseffekter på balansen

Om den obalanserade situation uppstod på grund av att elementet stoppades in i:

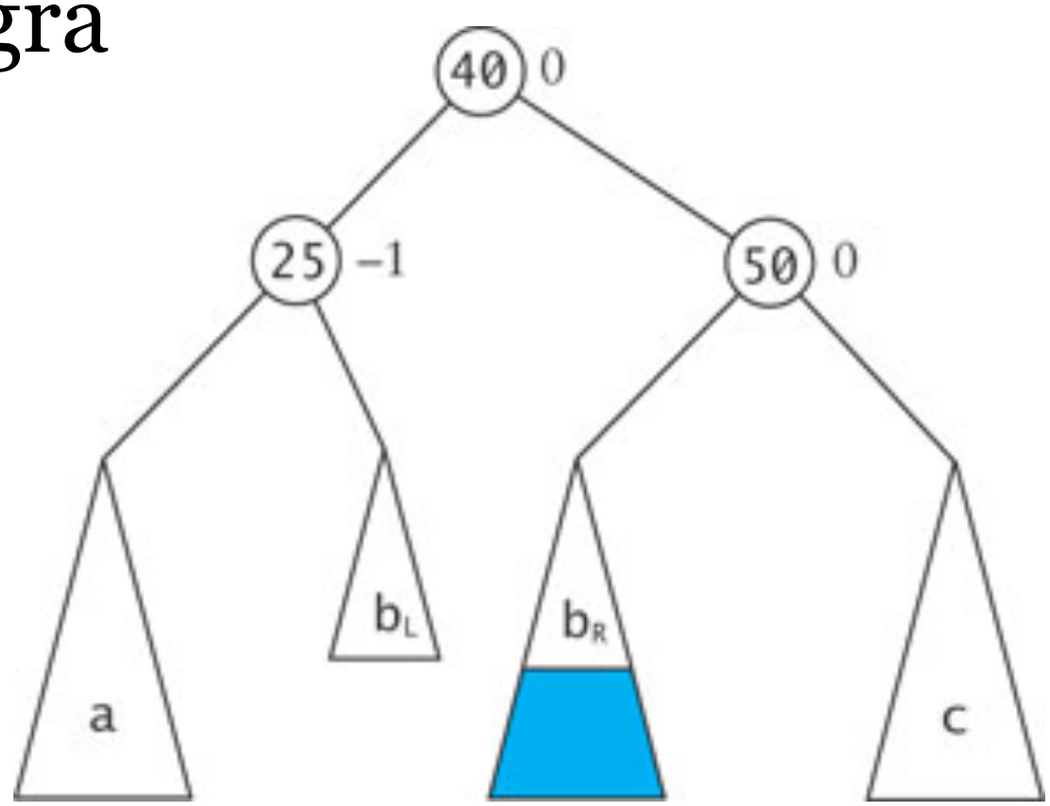
- delträd $\mathbf{b_L}$ (vänster-höger-vänster):
- roten och rotens vänstra barn har balansen 0 och balansen till vänstra barnet är +1



Rotationseffekter på balansen

Om den obalanserade situation uppstod på grund av att elementet stoppades in i:

- delträd $\mathbf{b_R}$ (vänster-höger-höger):
- roten och rotens högra barn har balansen 0 och balansen till vänstra barnet är –1



rebalanceLeft: revíderad version

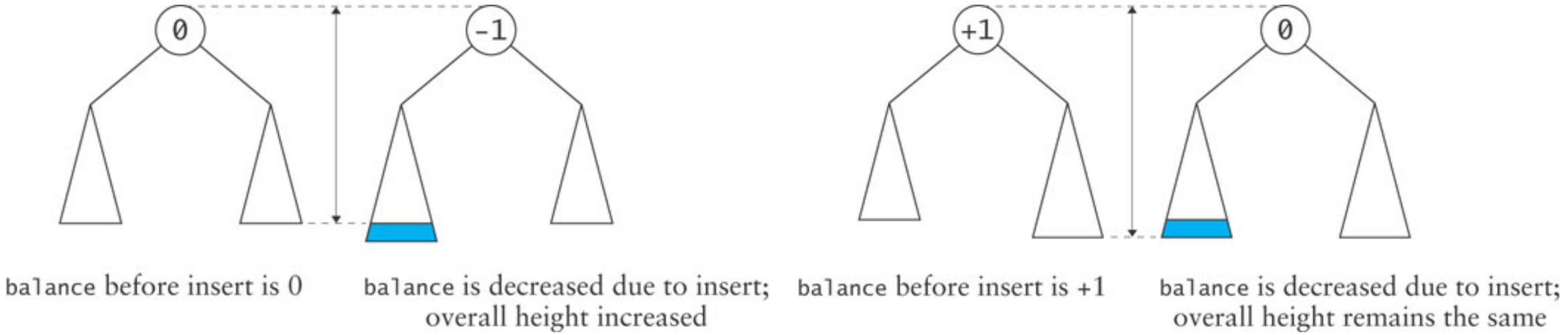
Revised Algorithm for rebalanceLeft

1. if the left subtree has a positive balance (Left-Right case):
2. 	if the left-left subtree has a negative balance (Left-Right-Left case):
3. 		Set the left subtree (new left subtree) balance to 0
4. 		Set the left-left subtree (new root) balance to 0
5. 		Set the local root (new right subtree) balance to +1
6. 	else (Left-Right-Right case):
7. 		Set the left subtree (new left subtree) balance to -1
8. 		Set the left-left subtree (new root) balance to 0
9. 		Set the local root (new right subtree) balance to 0
10. 	Rotate the left subtree left
11. else (Left-Left case):
12. 	Set the left subtree balance to 0
13. 	Set the local root balance to 0
14. Rotate the local root right

decrementBalance

När vi returnerar från en insättning i vänster delträd, behöver vi minska balansen

Vi behöver också kunna berätta för föräldrarna ifall delträdets höjd har ökat eller ej



balance before insert is 0 — balance is decreased due to insert; overall height increased

balance before insert is +1 — balance is decreased due to insert; overall height remains the same

Det finns två fall:

- noden är balanserad: insättning i vänstra delträdet kommer att göra noden vänster-tung, och höjden kommer att öka med 1
- noden är höger-tung: insättning i vänstra delträdet kommer att göra noden balanserad, och höjden kommer *inte* att öka

Borttagning från ett AVL-träd

Borttagning av noder är ungefär som insättning...

- ...fast precis tvärtom

Metoderna *decrementBalance*, *incrementBalance*, *rebalanceLeft* och *rebalanceRight* behöver modifieras så att de sätter variabeln *decrease* förutom variabeln *increase*

Effektivitet hos AVL-träd

AVL-träd är logaritmiska, O(log *n*), eftersom varje delträd är balanserat

- varje delträd får vara lite obalanserat (±1 balans), så trädet kan innehålla några hål
- i värsta fall (vilket är ovanligt) kan ett AVL-träd vara 1,44 gånger så högt som ett perfekt nodbalanserat träd
- empiriska tester visar att det i medeltal krävs 0,25+log(*n*) jämförelser för att sätta in element *n*
- eftersom vi kan ignorera konstanter så får vi en komplexitet på O(log *n*), även i praktiken