Hash tables (20.1-20.3, 20.5-20.6), AVL trees (19.4)

Hash tables naïvely

A hash table implements a set or map

The plan: take an array of size k

Define a *hash function* that maps values to indices {0,...,k-1}

To find, insert or remove a value,
compute the hash
and put it at that index

Hash tables naïvely, example

Suppose the values are integers, and take an array of size 5 and a hash function $h(n) = n \bmod 5$

0	1	2	3	4
5		17	8	

This hash table contains {5, 8, 17}

Inserting 14 gives:

0	1	2	3	4
5		17	8	**14**

Hash tables naïvely, example

Suppose the values are integers, and take an array of size 5 and a hash function $h(n) = n \bmod 5$

0	1	2	3	4
5		17	8	

This hash table contains {5, 8, 17}

Inserting 12 gives... what???

0	1	2	3	4
5		17	8	

12 should go in index 2, but there is already a value here – a *collision*

The problem with naïve hash tables

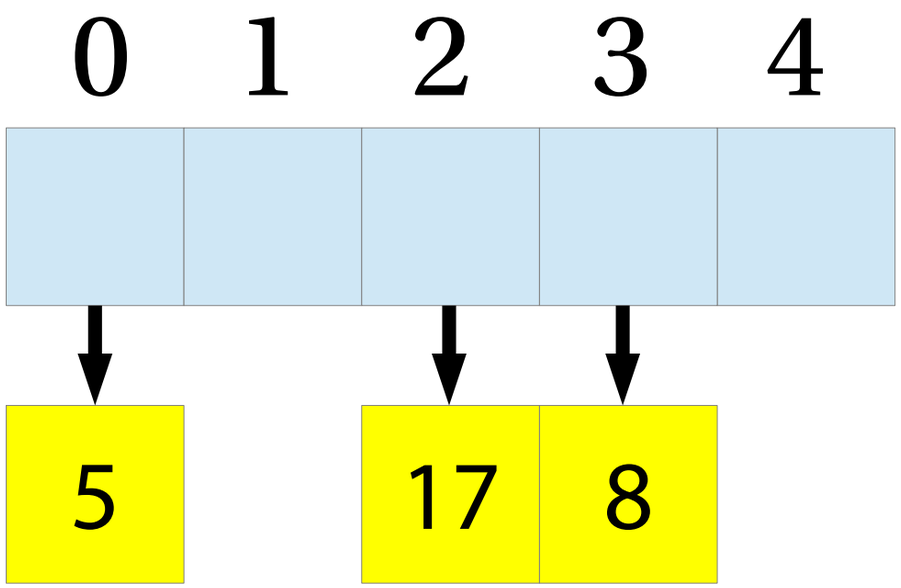
Naïve hash tables have two problems:

1. Sometimes two values have the same hash – this is called a *collision*
   - Two ways of avoiding collisions, *chaining* and *probing* – we will see them later
2. The hash function is specific to a particular size of array
   - Allow the hash function to return an arbitrary integer and then take it modulo the array size: *h(x) = x.hashCode() mod array.size*

Avoiding collisions: chaining

Instead of an array of elements, have an array of *linked lists*

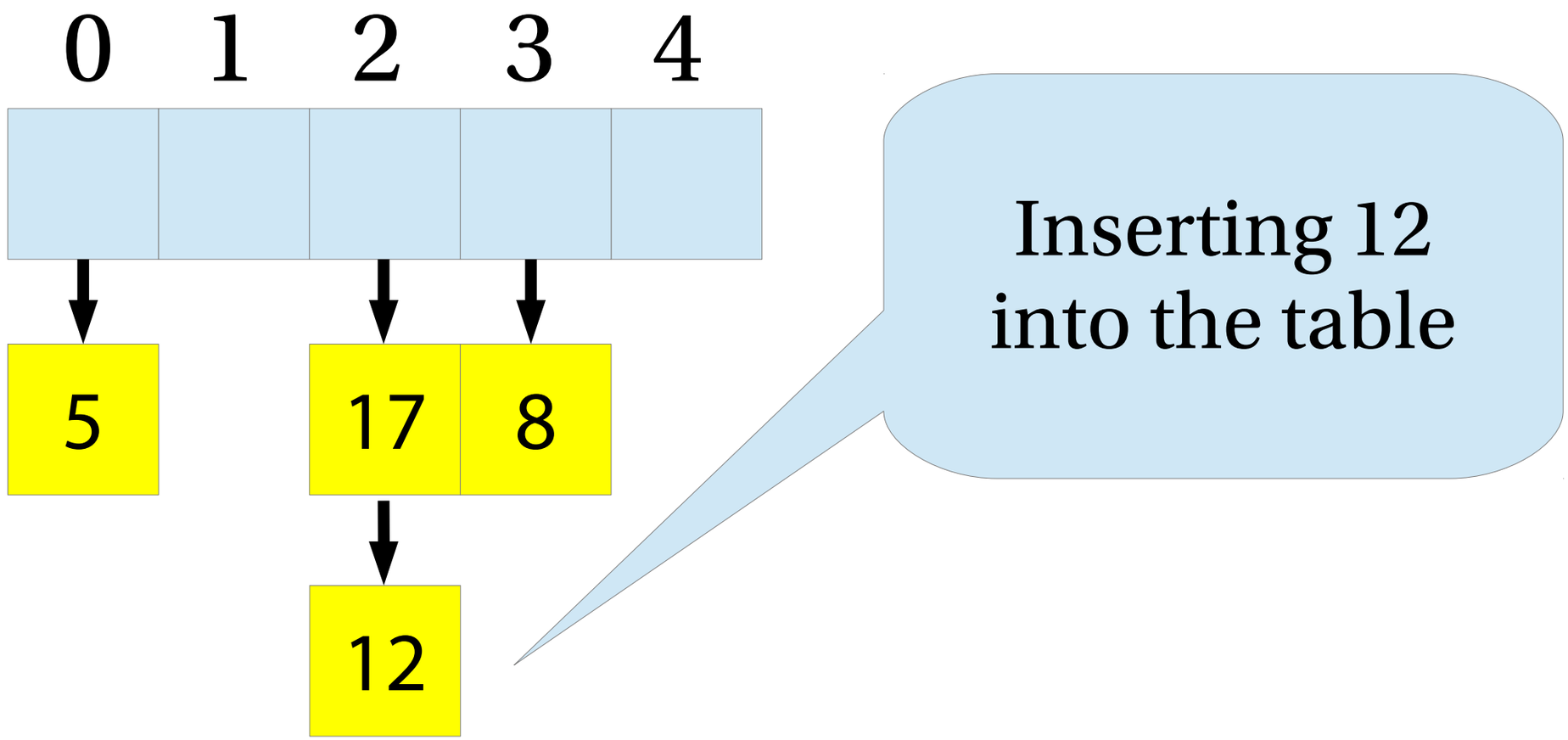
To add an element, calculate its hash and insert it into the list at that index



Avoiding collisions: chaining

Instead of an array of elements, have an array of *linked lists*

To add an element, calculate its hash and insert it into the list at that index



Performance of chained hash tables

If the linked lists are small, chained hash tables are fast

- If the size is bounded, operations are O(1) time

But if they get big, everything gets slow

Observation 1: the array must be big enough

- If the hash table gets too full (a high *load factor*), allocate a new array of twice the size (*rehashing*)

Observation 2: the hash function must *evenly distribute* the elements!

- If everything gets the same hash, all operations are O(n)

Defining a good hash function

What is wrong with the following hash function on strings?

Add together the character code of each character in the string
(character code of a = 97, b = 98, c = 99 etc.)

Maps e.g. *bass* and *bart* to the same hash code! (s + s = r + t)

Similar strings will be mapped to nearby hash codes – does not distribute strings evenly

A hash function on strings

An idea: map strings to integers as follows:

$$s_0 \cdot 128^{n-1} + s_1 \cdot 128^{n-2} + \ldots + s_{n-1}$$

where s_i is the code of the character at index i

If all characters are ASCII (character code 0 – 127), each string is mapped to a different integer!

The problem

In many languages, when calculating

$$s_0 \cdot 128^{n-1} + s_1 \cdot 128^{n-2} + \ldots + s_{n-1},$$

the calculation happens modulo 2^{32} (*integer overflow*)

So the hash will only use the last few characters!

Solution: replace 128 with 37

$$s_0 \cdot 37^{n-1} + s_1 \cdot 37^{n-2} + \ldots + s_{n-1}$$

Use a *prime number* to get a good distribution

This is what Java uses for strings

Hashing a pair

```
class C { A a; B b; }
```

One way: multiply the two hash codes by different prime numbers and add the results, then add a constant:

```
int hashCode() {
    return 31 * a.hashCode() +
           37 * b.hashCode() + 1;
}
```

Hash functions

A good hash function must distribute elements evenly to avoid collisions

Defining really good hash functions is a black art – but the two techniques above give you decent hash functions

Making the array size a prime number helps mask patterns in the hash function

- e.g., if the hash function always returns an even number, if the array size is a power of two then all the odd indexes will be empty

Linear probing

Another way of dealing with collisions is *linear probing*

Uses an array of values, like in the naïve hash table

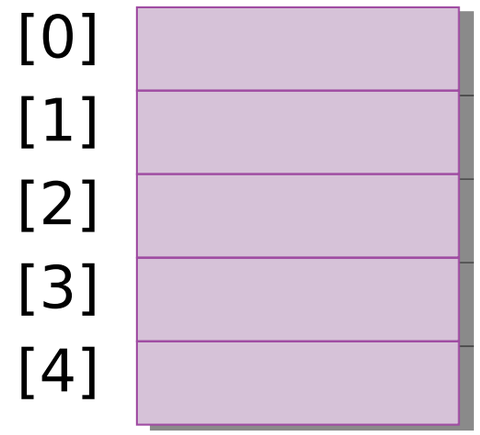
If you want to store a value at index *i* but it's full, store it in index *i+1* instead!

If that's full, try *i+2*, and so on

...if you get to the end of the array, wrap around to 0

Exempel: Insättning

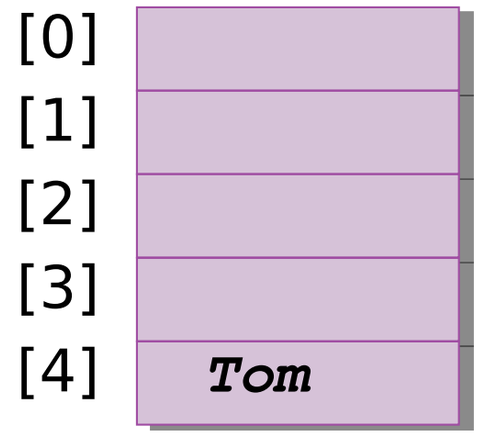
Tom Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

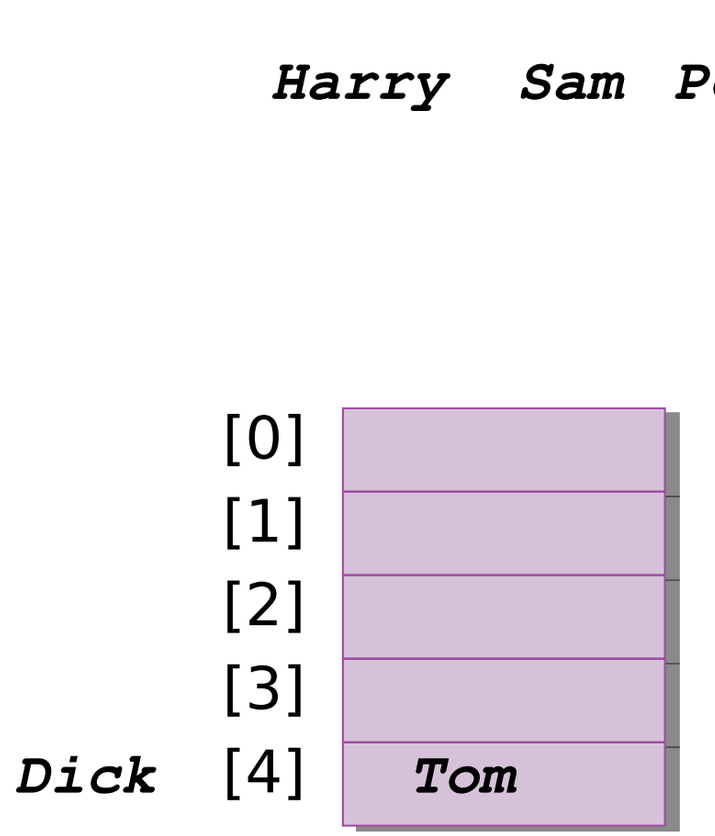
Exempel: Insättning

Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

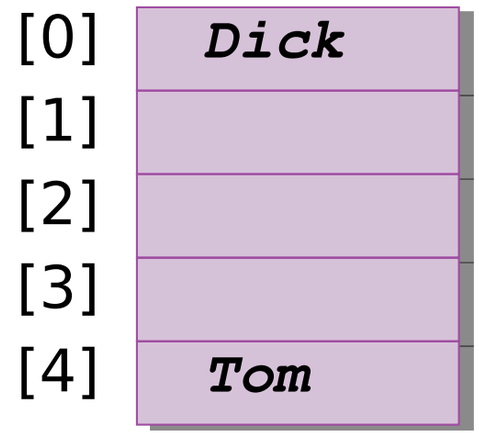
Exempel: Insättning



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

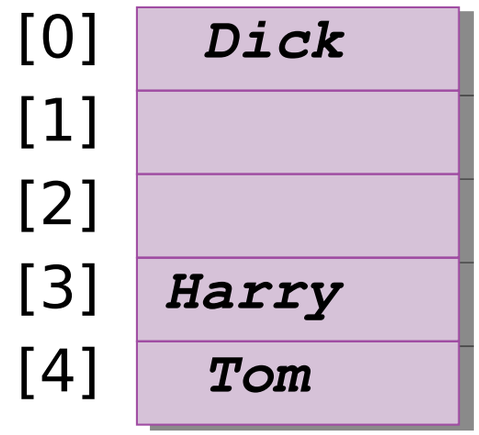
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

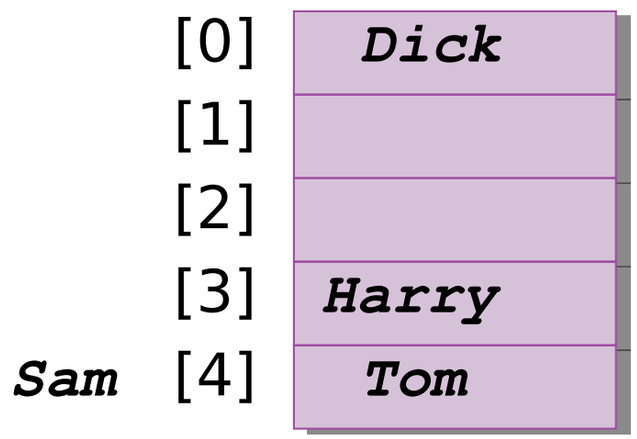
Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete

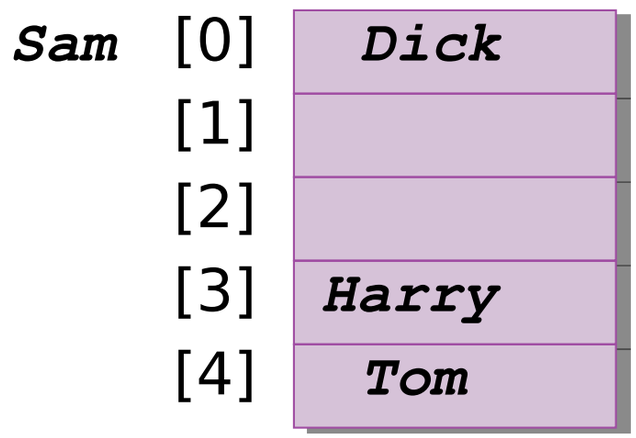


Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

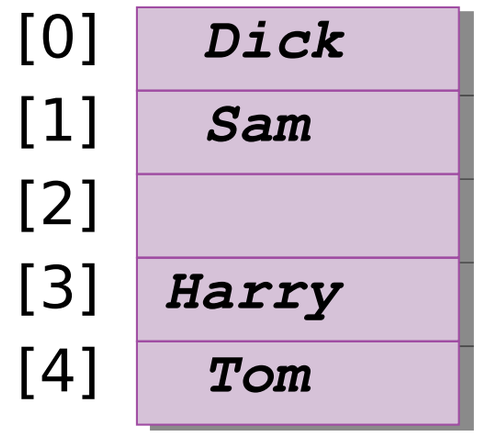
Pete

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



Exempel: Insättning

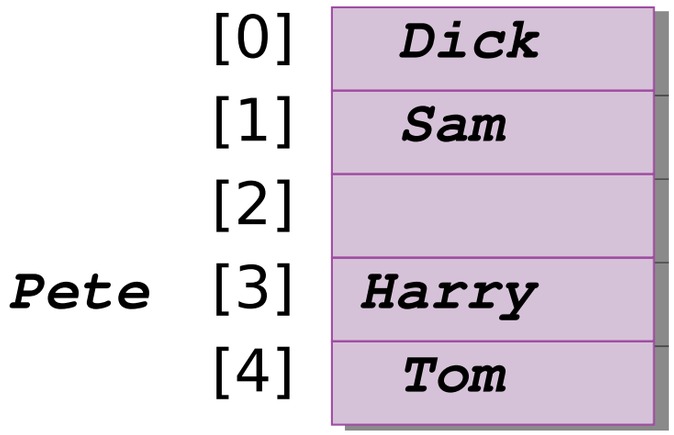
Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

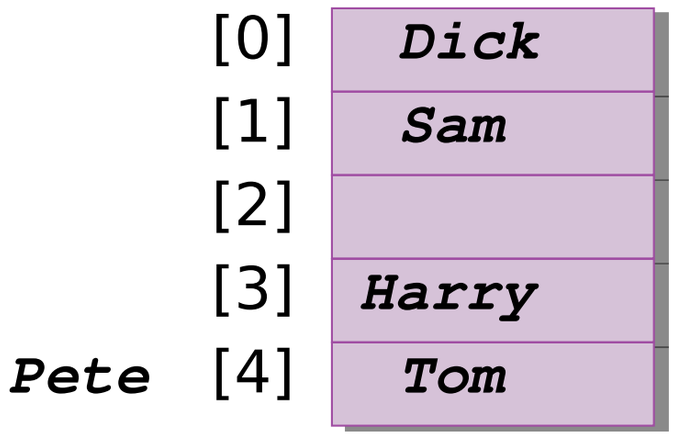
Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



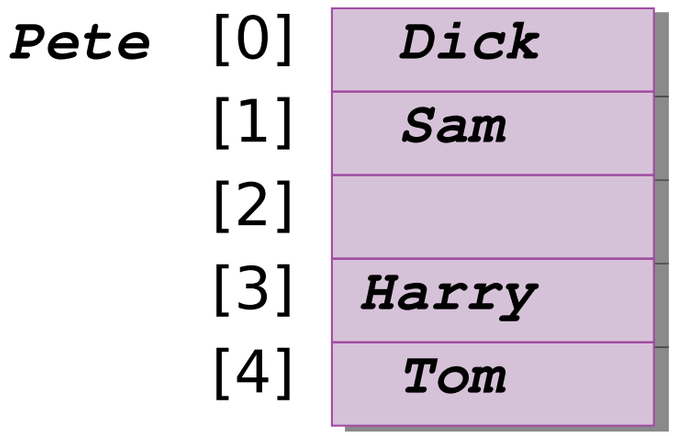
Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



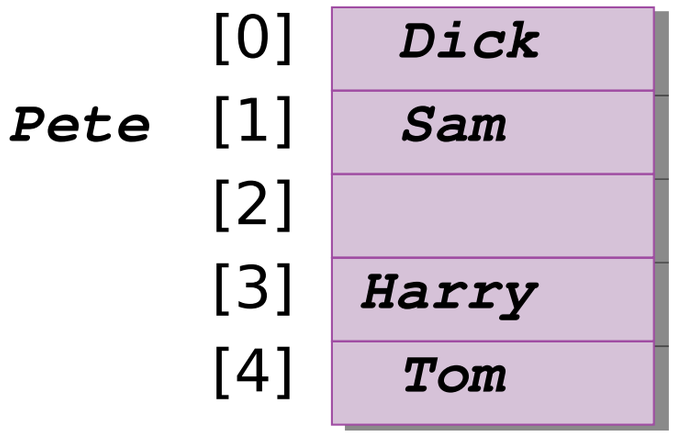
Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



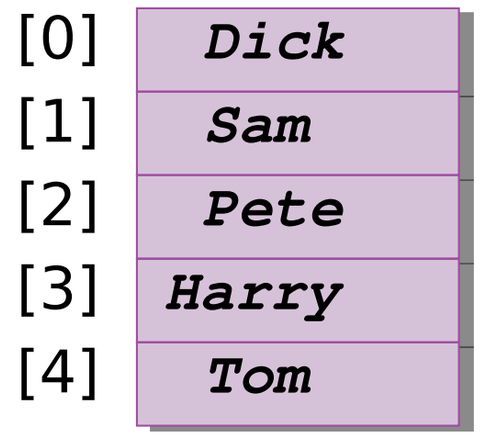
Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



Exempel: Insättning

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



To find “Pete” (hash 3), you must start at index 3 and work your way all the way around to index 2

Searching with linear probing

To find an element under linear probing:

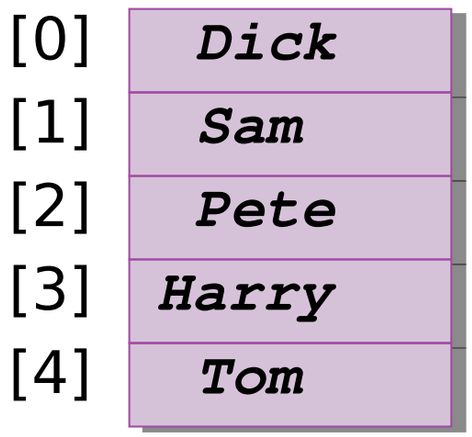
- Calculate the hash of the element, *i*
- Look at *array[i]*
- If it's the right element, return it!
- If there's no element there, fail
- If there's a *different* element there, search again at index *(i+1) % array.size*

We call a group of adjacent non-empty indices a *cluster*

Deleting with linear probing

Can't just remove the element...

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

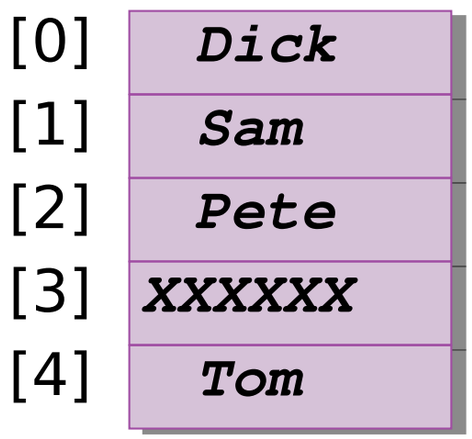


If we remove Harry,
Pete will be in the wrong cluster
and we won't be able to find him

Deleting with linear probing

Instead, mark it as deleted (*lazy deletion*)

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



The search algorithm will skip over XXXXXX

Deleting with linear probing

It's useful to think of the invariant here:

- Linear *chaining*: each element is found at the index given by its hash code
- Linear *probing*: each element is found at the index given by its hash code, *or a later index in the same cluster*

Naïve deletion will split a cluster in two, which may break the invariant

Hence the need for an empty value that does not mark the end of a cluster

Linear probing performance

To insert or find an element under linear probing, you might have to look through a whole cluster of elements

Performance depends on the size of these clusters:

- Small clusters – expected O(1) performance
- Almost-full array – O(n) performance
- If the array is full, you can't insert anything!

Thus you need:

- to expand the array and *rehash* when it starts getting full
- a hash function that distributes elements evenly

Same situation as with linear chaining!

Linear probing vs linear chaining

In linear chaining, if you insert several elements with the same hash *i*, those elements become slower to find

In linear probing, elements with hash *i+1*, *i+2*, etc., will belong to the same cluster as element *i*, and will also get slower to find

If the load factor is too high, this tends to result in very long clusters in the hash table – a phenomenon called *primary clustering*

Probing vs chaining

Linear probing is more sensitive to high load

On the other hand, linear probing uses less memory for a given load factor, so you can use a bigger array than you would with chaining

load factor (#elements / array size)	**number of comparisons (linear probing)**	**number of comparisons (linear chaining)**
0 %	1.00	1.00
25 %	1.17	1.13
50 %	1.50	1.25
75 %	**2.50**	1.38
85 %	3.83	1.43
90 %	5.50	1.45
95 %	10.50	1.48
100 %	—	1.50
200 %	—	2.00
300 %	—	**2.50**

Summary of hash table design

Several details to consider:

- *Rehashing*: resize the array when the load factor is too high
- *A good hash function*: need an even distribution
- *Collisions*: either chaining or probing

Hash tables have *expected* (average) O(1) performance if the hash function is random (there are no patterns) – but it's normally not!

Nevertheless, performance is O(1) in practice with decent hash functions.

So – theoretical foundations a little shaky, but very good practical performance.

Hash tables versus BSTs

Hash tables: O(1) performance in practice (O(n) if very unlucky), BSTs: O(log n) if balanced

Hash tables are *unordered*: you can't e.g. get the elements in increasing order

But they are normally *faster* than balanced BSTs, despite the theoretical O(n) worst case

AVL trees (19.4)

(many slides taken from Peter Ljunglöf)

A reminder

Insertion, deletion and search in a binary search tree take O(height of tree) time

Thus, if the tree is balanced, it's O(log n); if unbalanced, it's O(n)

Balanced BSTs make sure the height of the tree is always O(log n)

Insert/delete work as in a BST but do some extra work to rebalance the tree

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

AVL trees

The AVL tree is the first balanced BST discovered (from 1962) – it's named after Adelson-Velsky and Landis

It's a BST with the following invariant:

- The *difference in heights* between the left and right children of any node is at most 1

This makes the tree's height O(log n)

AVL trees

We call the quantity *right height – left height* the *balance* of a node

Thus the AVL invariant is: the balance of every node is -1, 0, or 1

Whenever a node gets out of balance, we fix it with *tree rotations*

Implementation: store the balance of each node as a field in the node, and remember to update it when rotating the tree

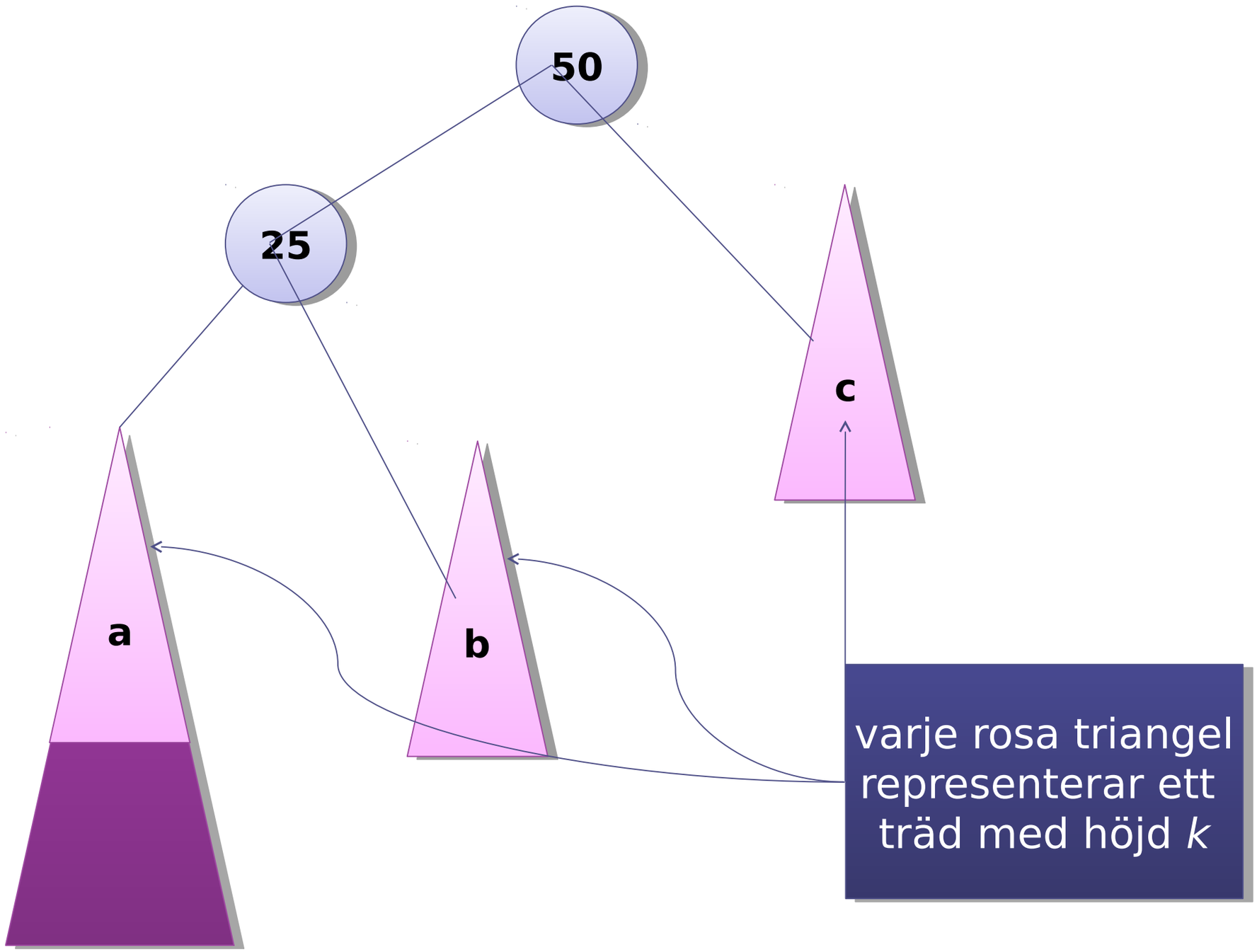
AVL insertion

Start by doing a BST insertion

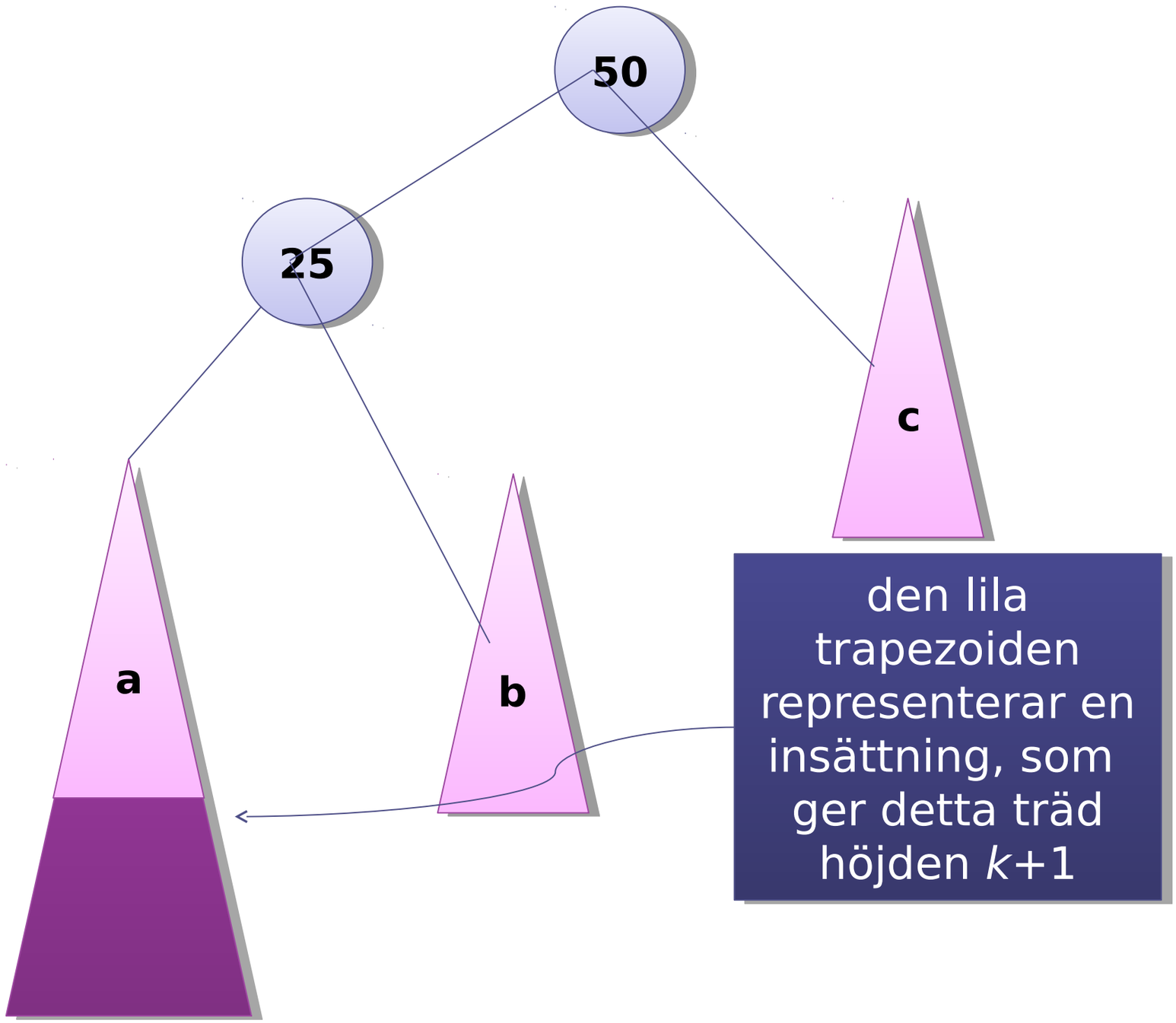
Then go upwards from the newly-inserted node, rotating unbalanced nodes as you go

There are several cases depending on *how* the tree is unbalanced

Balansera ett vänster-vänsterträd

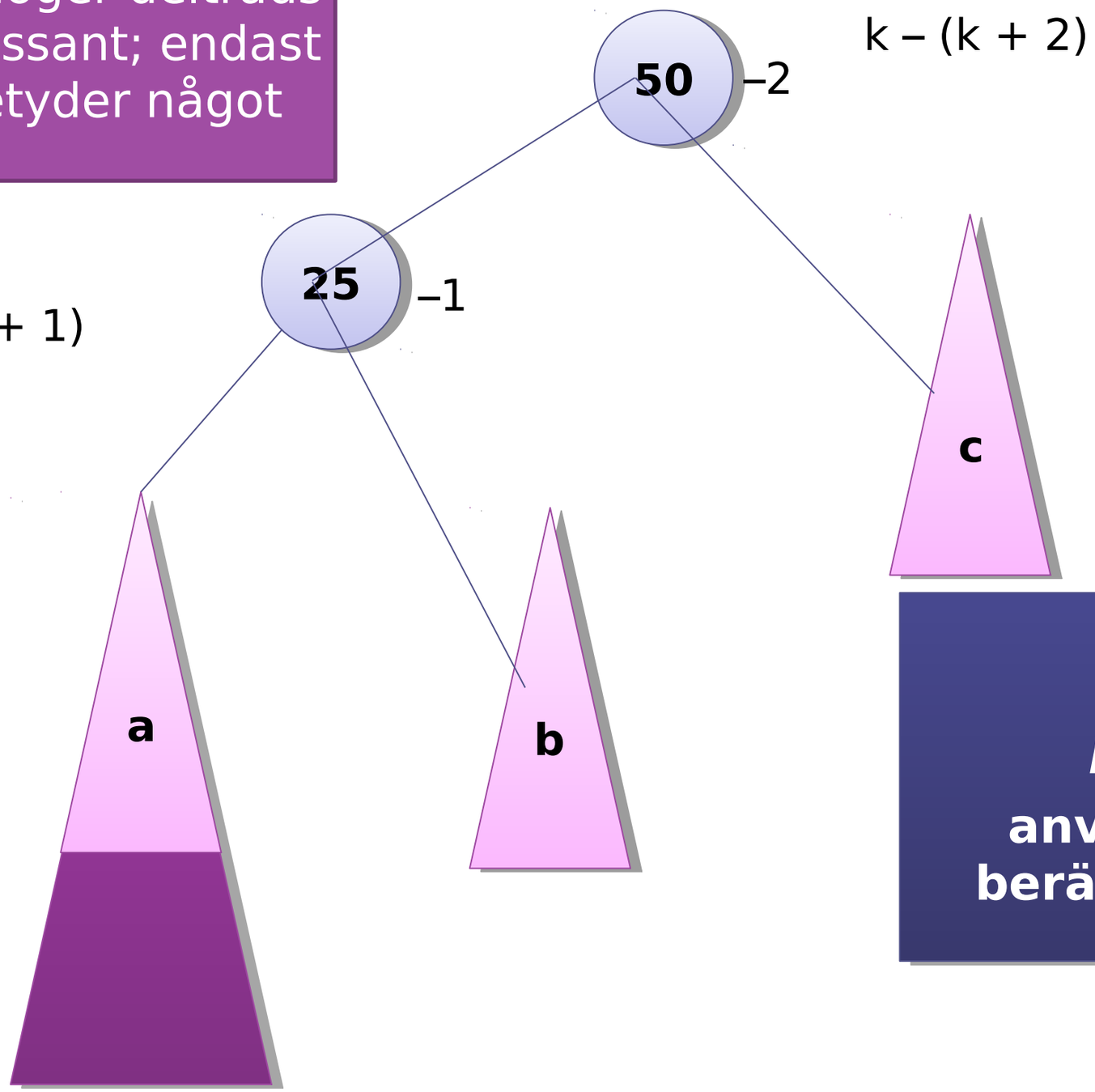


Balansera ett vänster-vänsterträd



Balansera ett vänster-vänsterträd

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

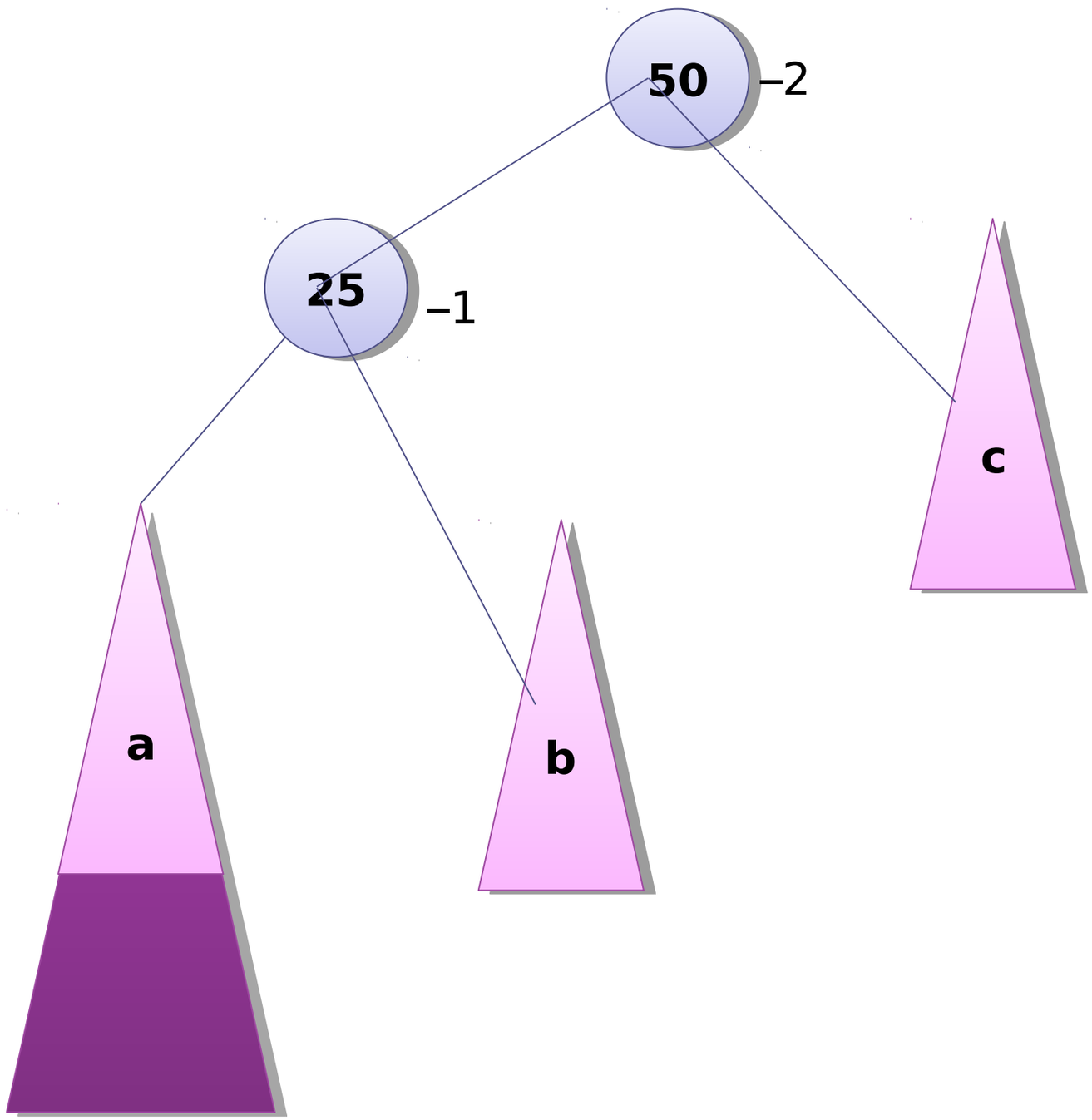


formeln

$$h_R - h_L$$

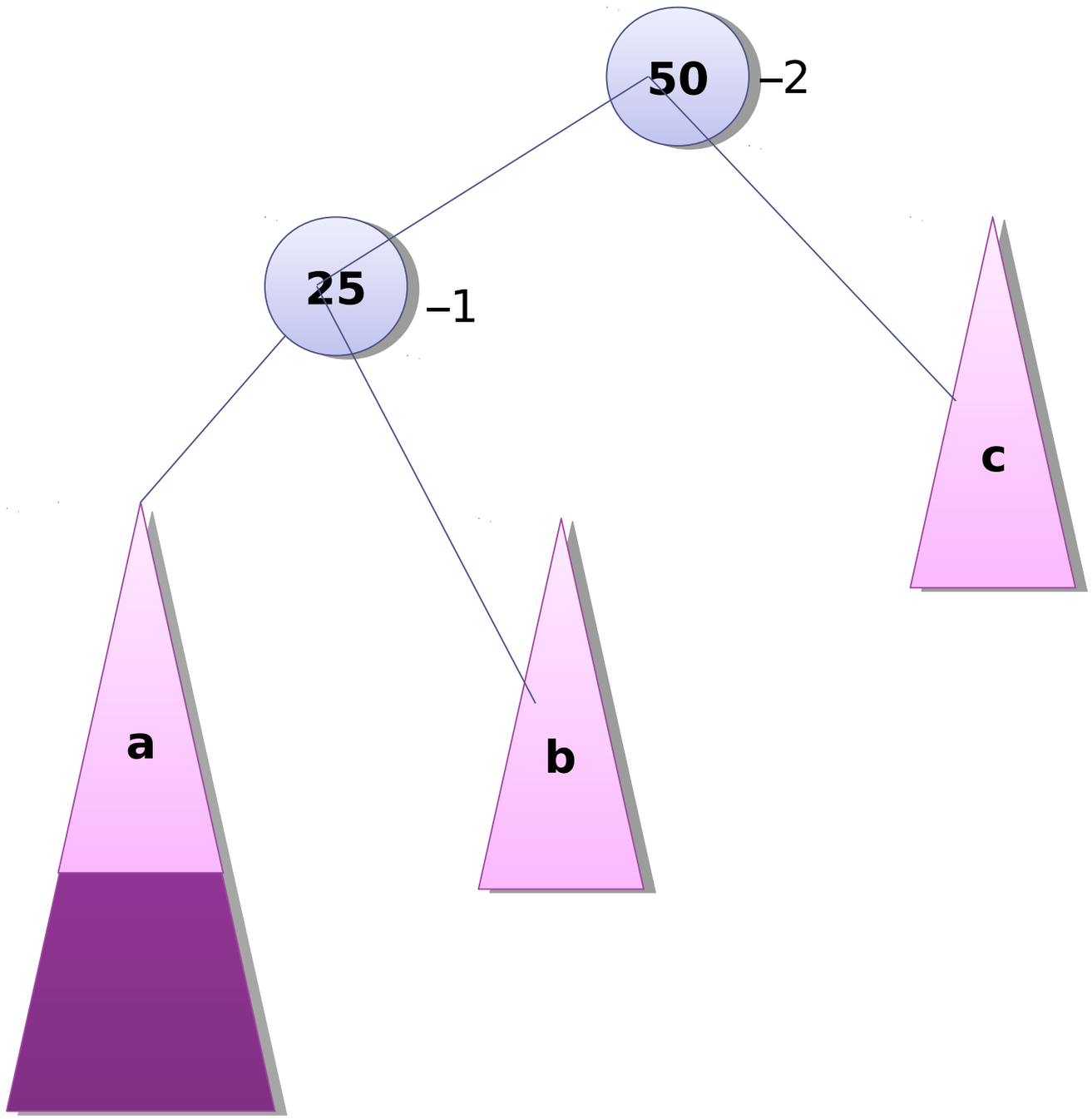
används för att beräkna balansen

Balansera ett vänster-vänsterträd



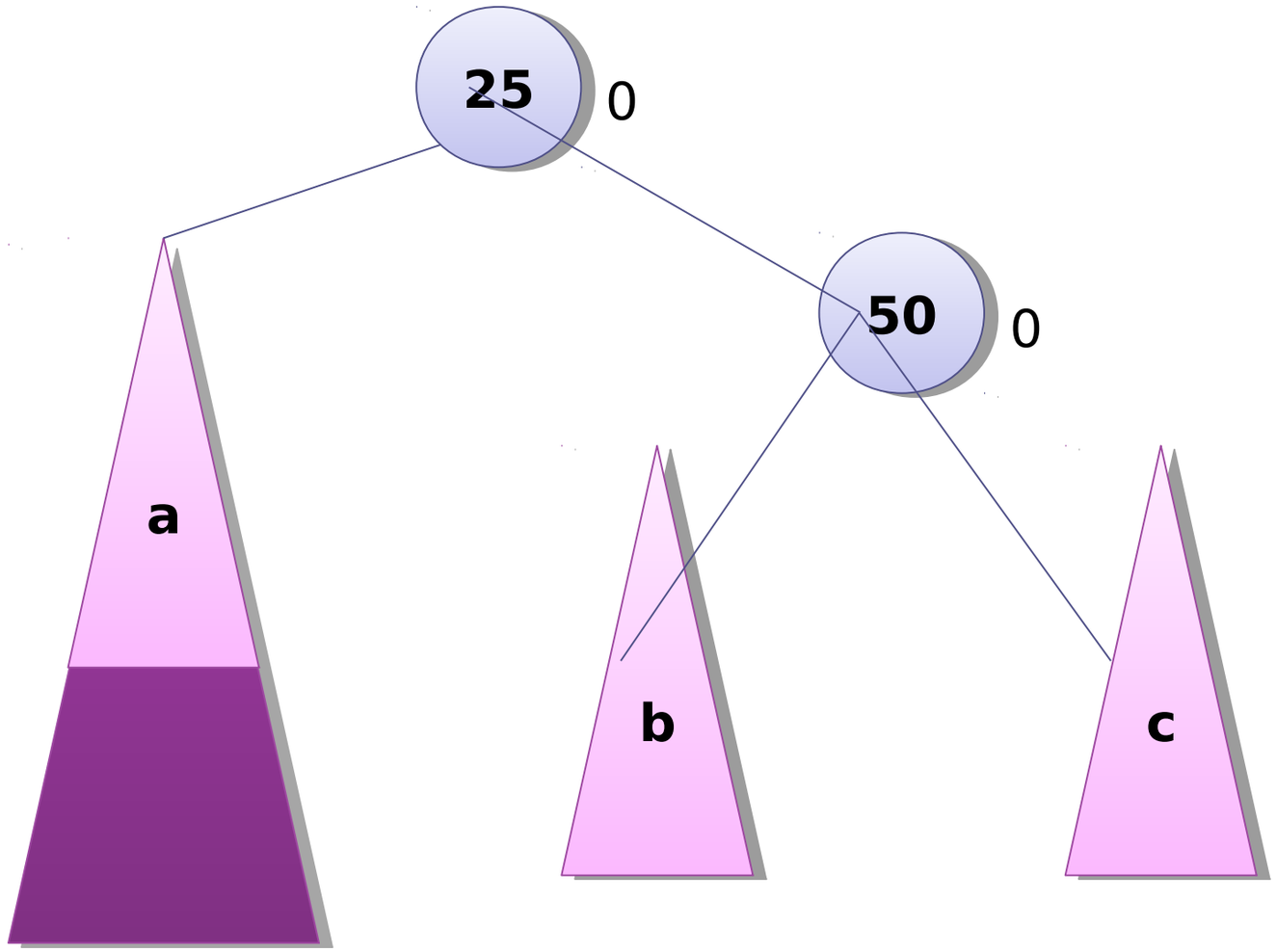
när både roten och vänster delträd ”lutar åt vänster”, så kallas trädet ett vänster-vänsterträd

Balansera ett vänster-vänsterträd



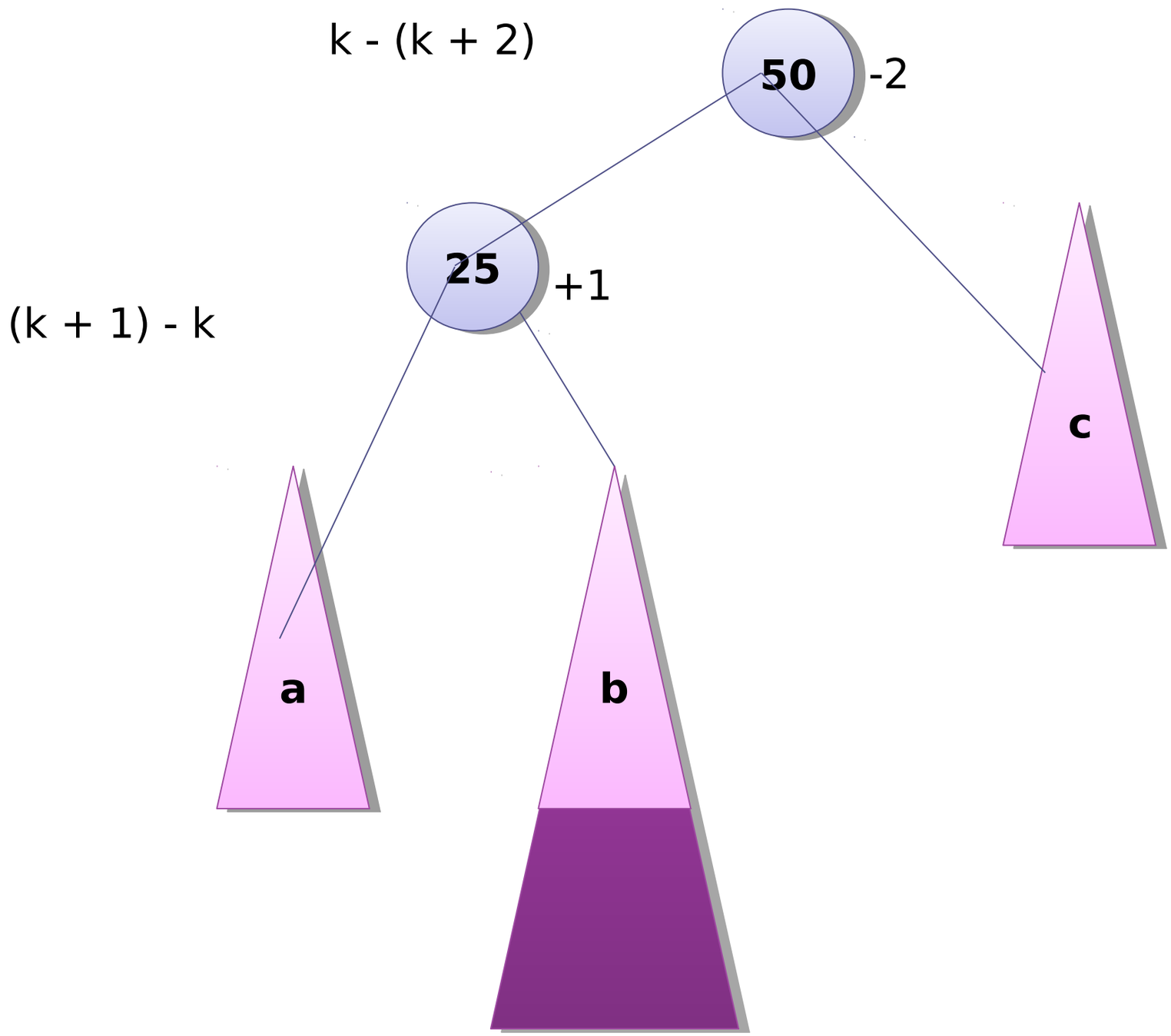
vänster-vänsterträd kan balanseras genom att rotera åt höger

Balansera ett vänster-vänsterträd

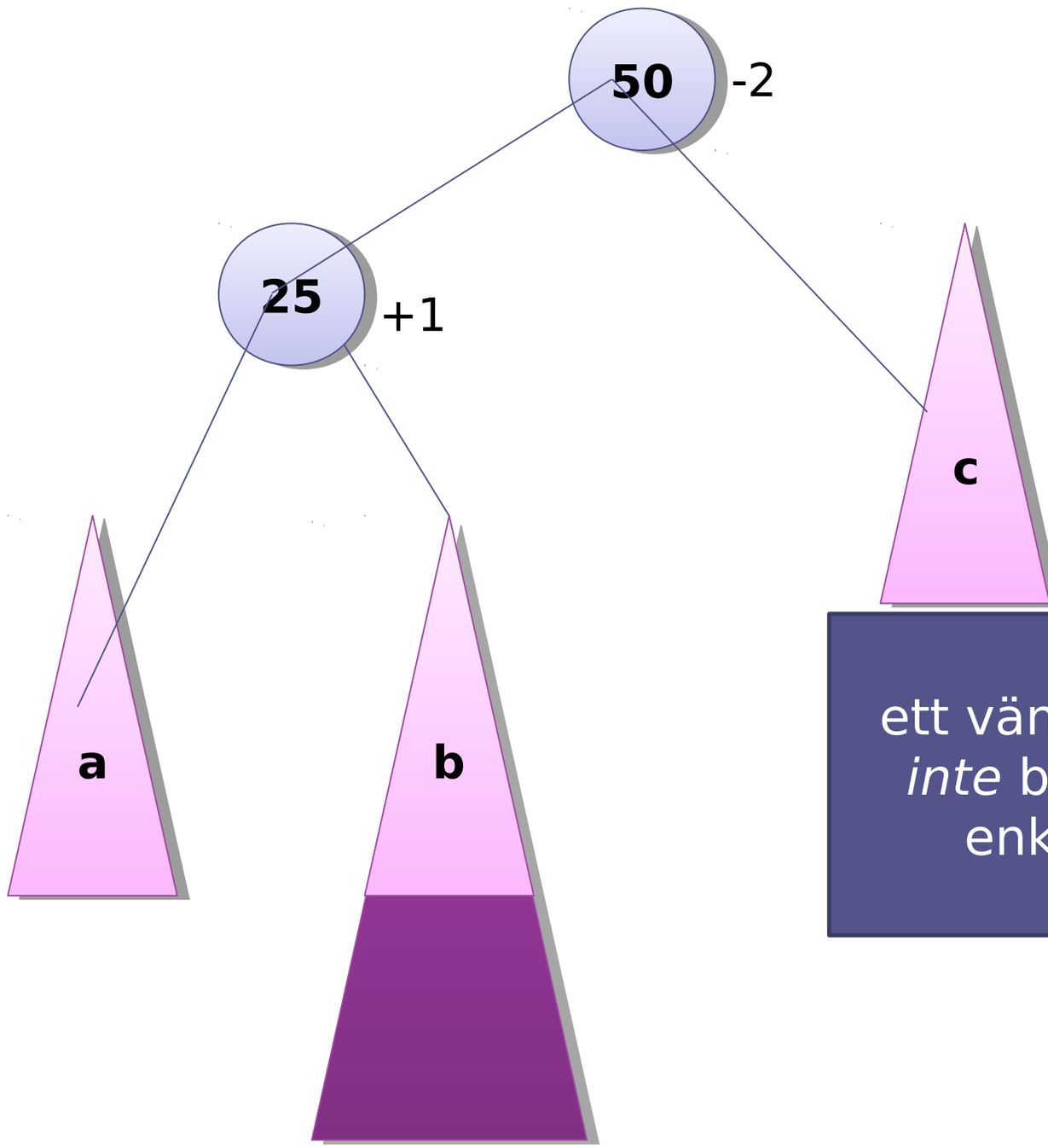


notera att den totala höjden *inte* har ändrats jämfört med innan insättningen!

Balansera ett vänster-högerträd

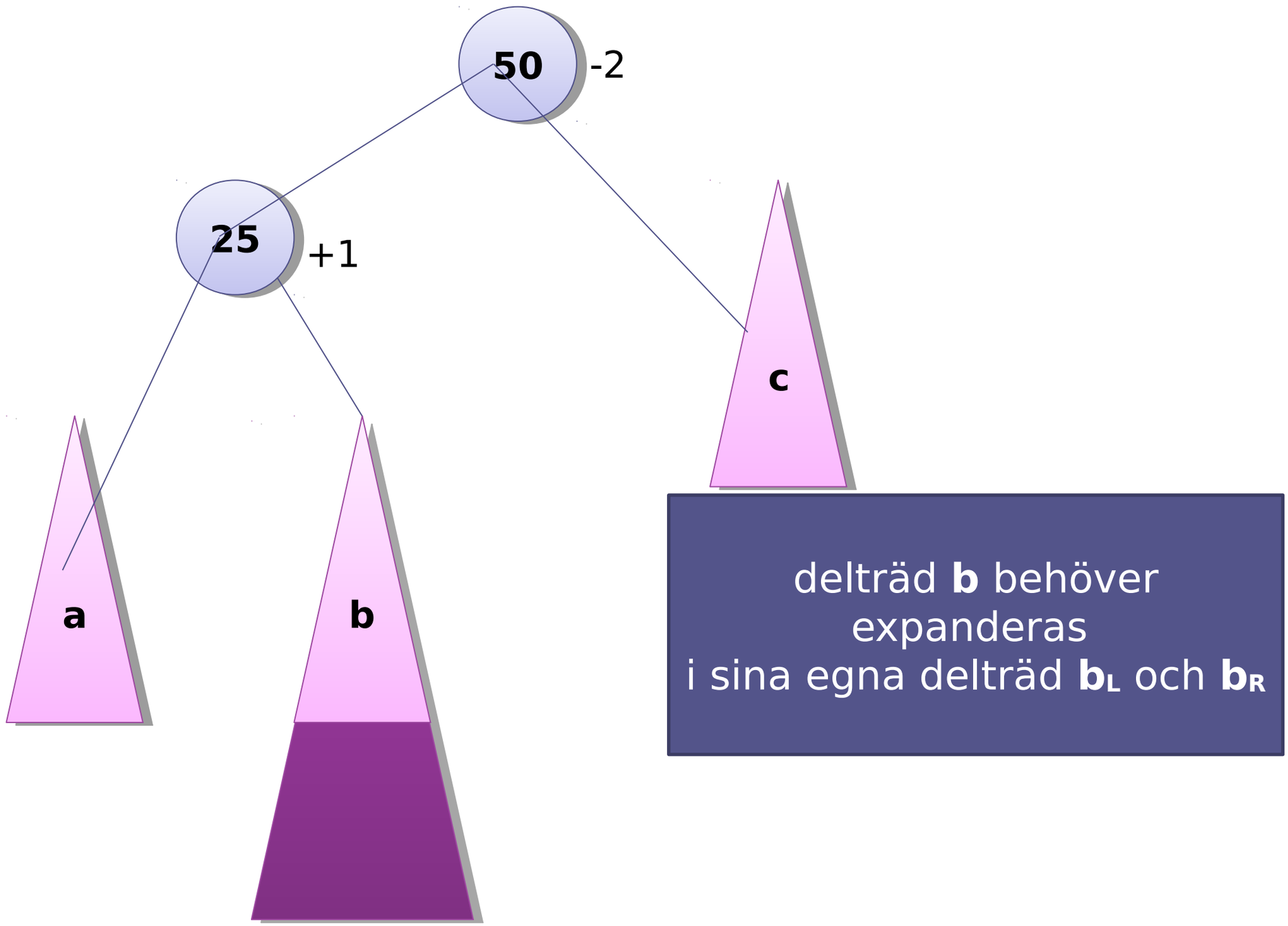


Balansera ett vänster-högerträd



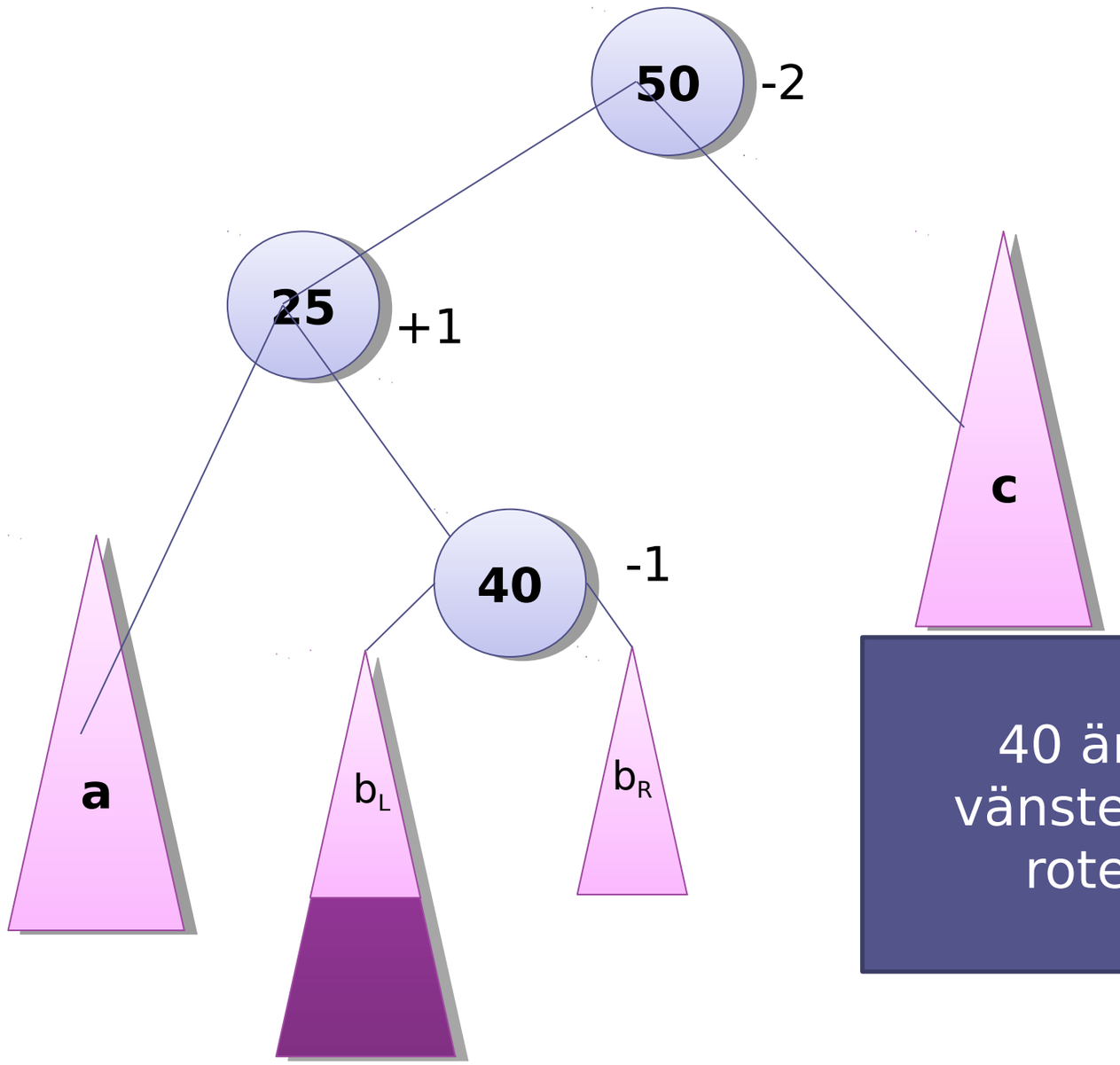
ett vänster-högerträd kan *inte* balanseras med en enkel högerrotation

Balansera ett vänster-högerträd



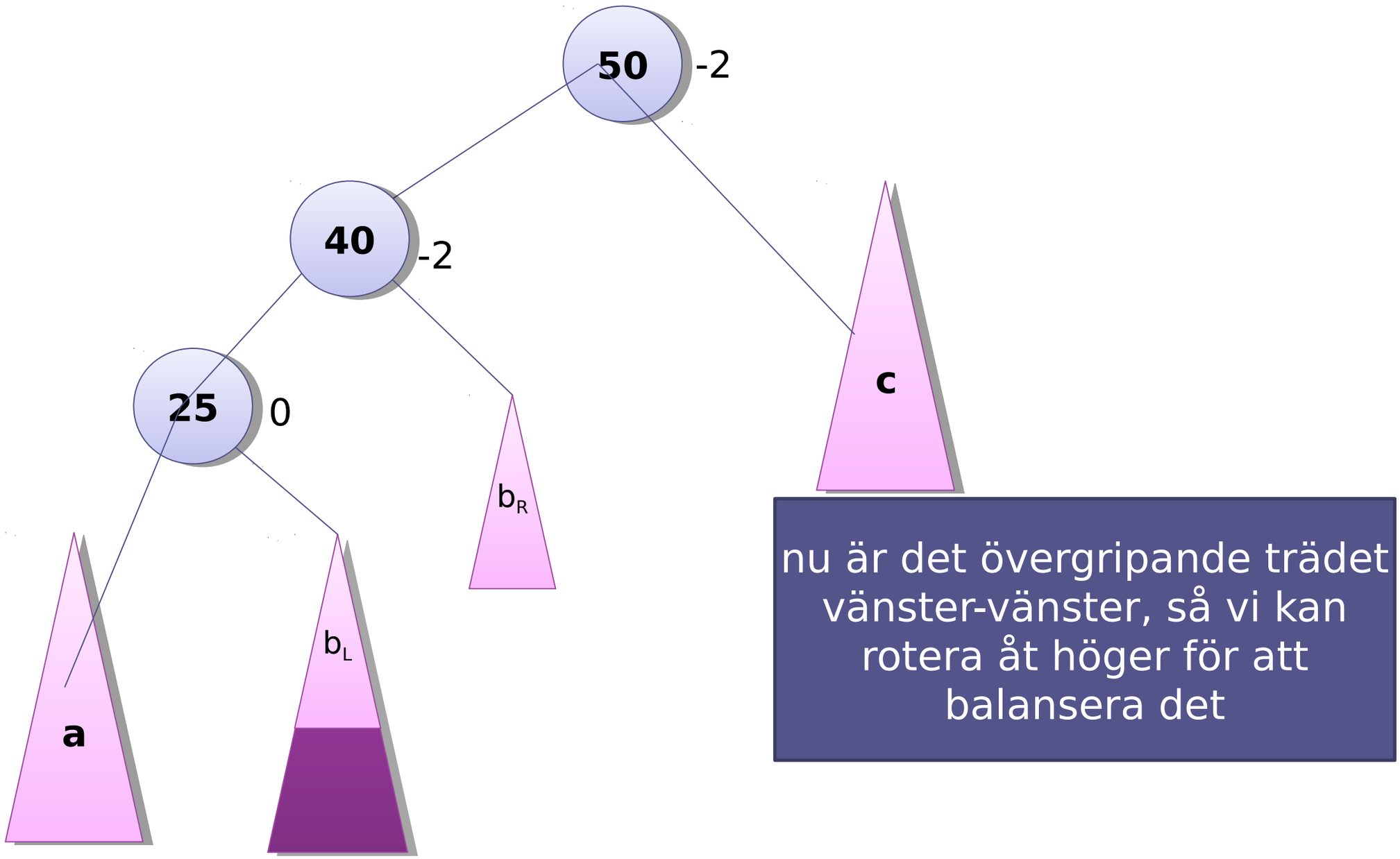
delträd **b** behöver
expanderas
i sina egna delträd $\mathbf{b_L}$ och $\mathbf{b_R}$

Balansera ett vänster-högerträd



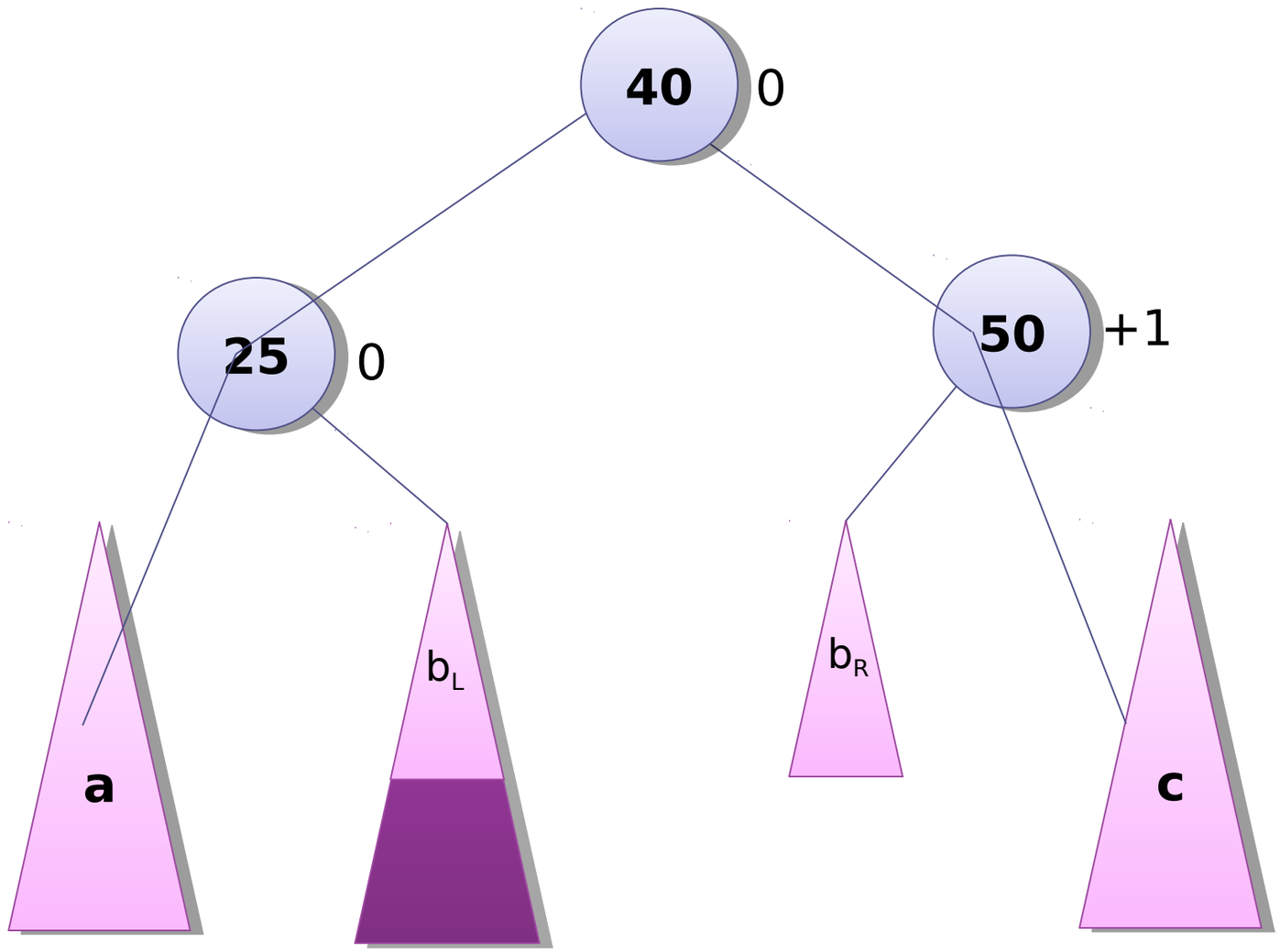
40 är vänster-tungt: vänster delträd kan nu roteras åt vänster

Balansera ett vänster-högerträd

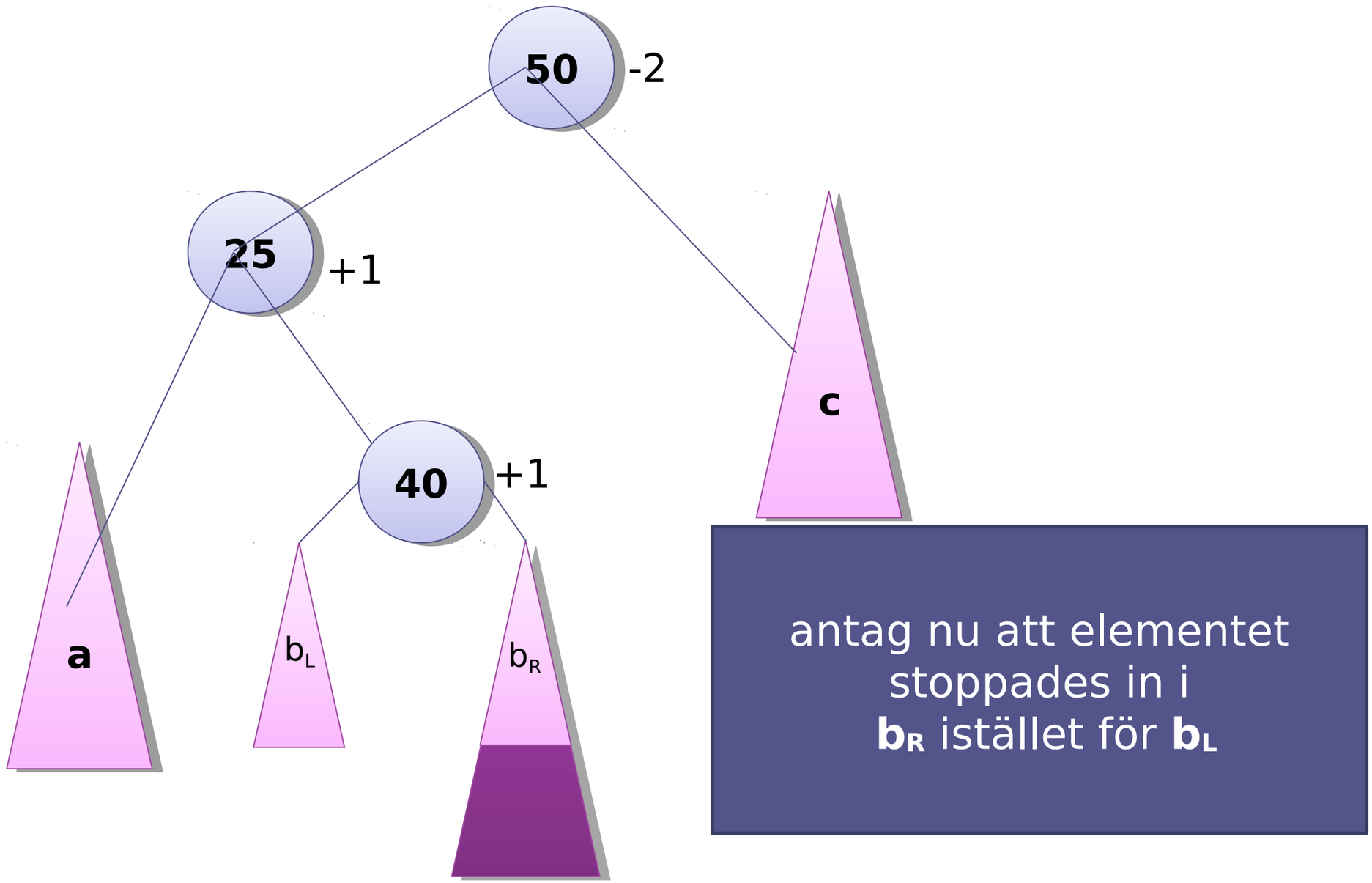


nu är det övergripande trädet vänster-vänster, så vi kan rotera åt höger för att balansera det

Balansera ett vänster-högerträd

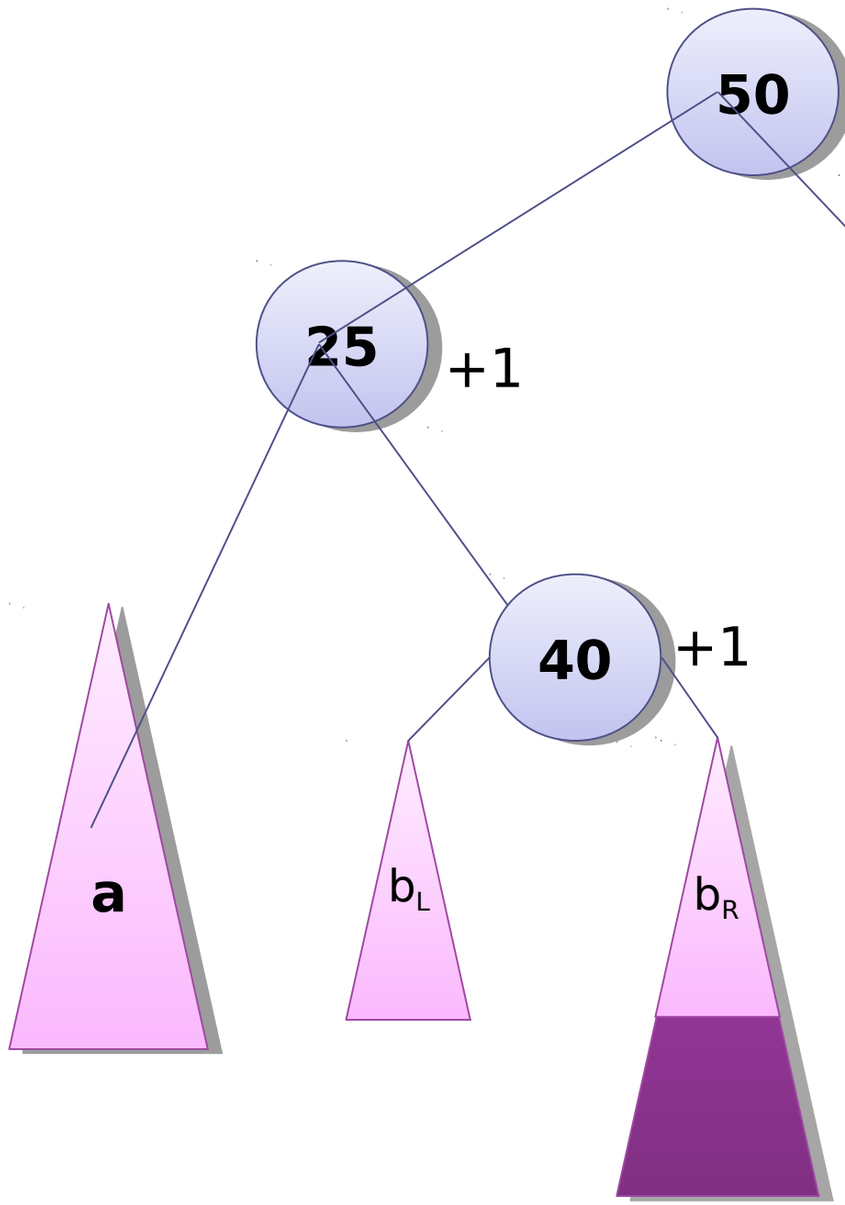


Balansera ett vänster-högerträd



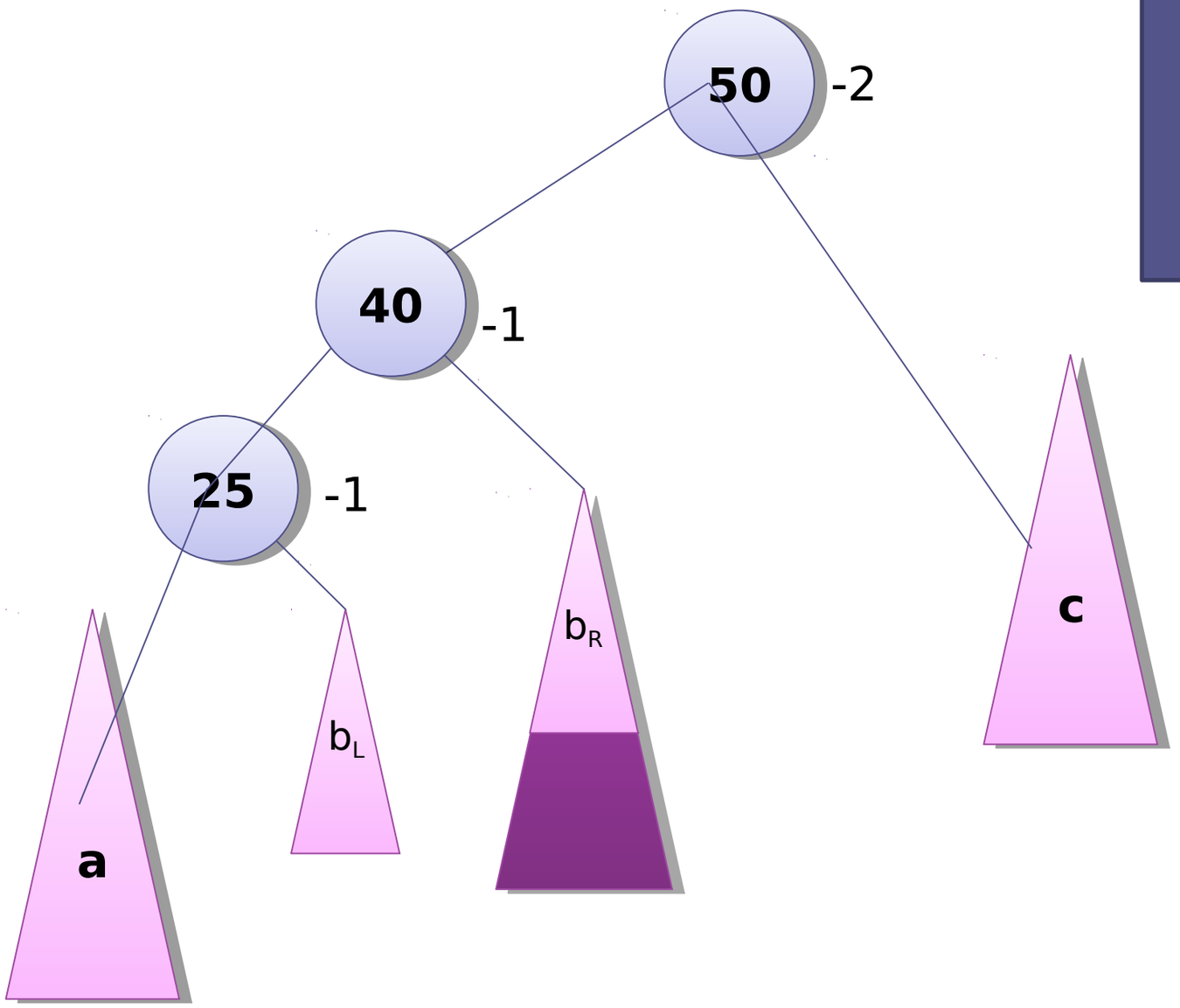
antag nu att elementet stoppades in i **b_R** istället för **b_L**

Balansera ett vänster-högerträd



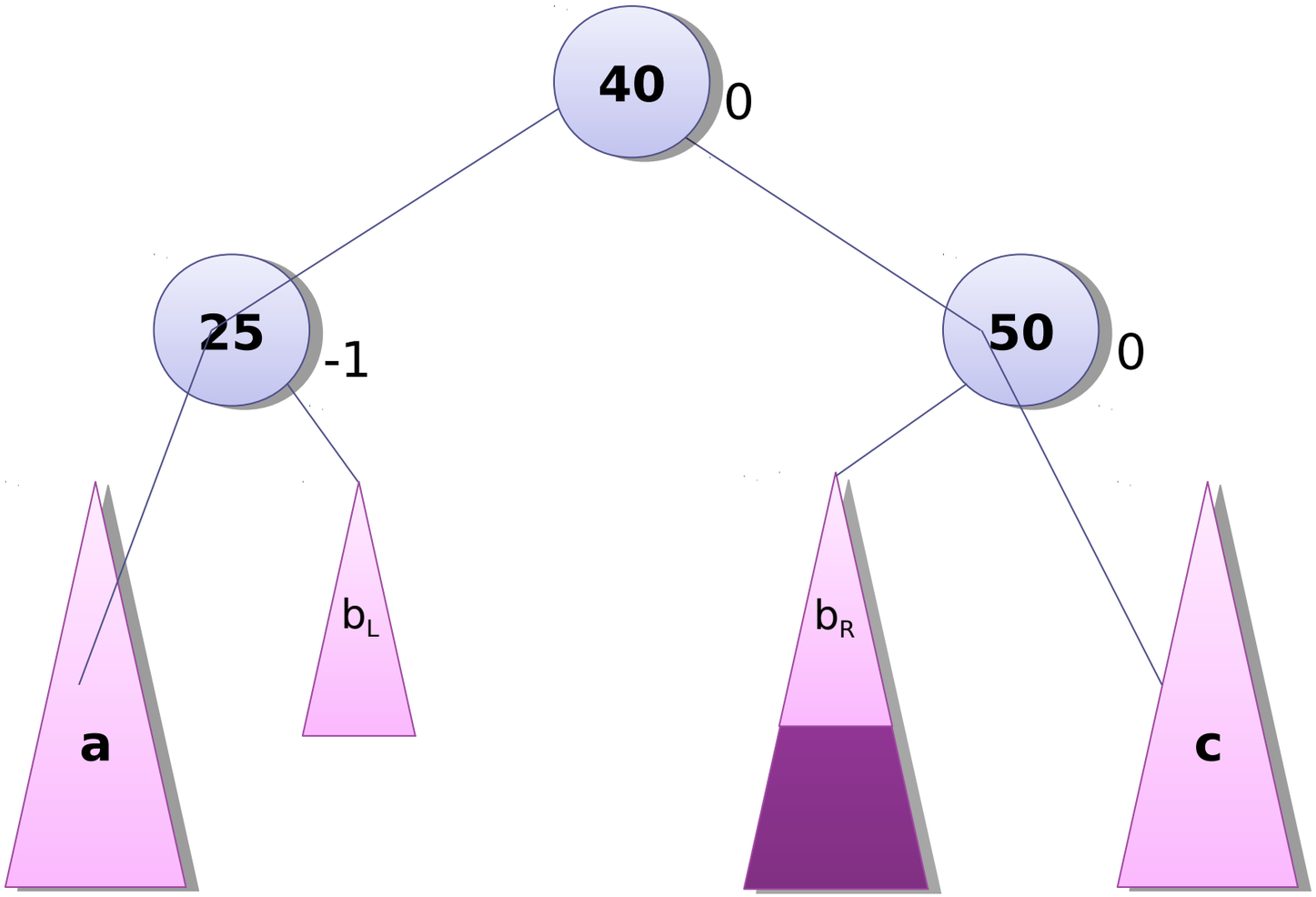
1. rotera vänster delträd åt vänster

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

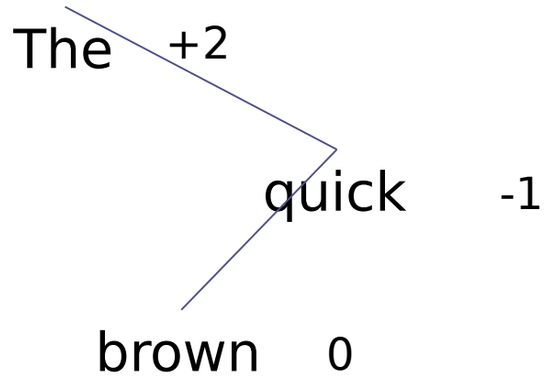
(bilden är från Wikipedia)

Ett större AVL-exempel

Nu ska vi bygga ett AVL-träd för orden i

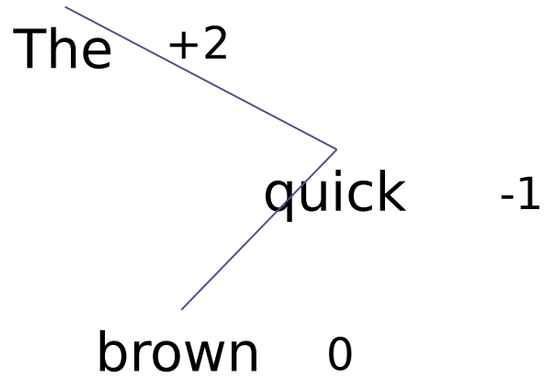
”The quick brown fox jumps over the lazy dog”

The quick brown...



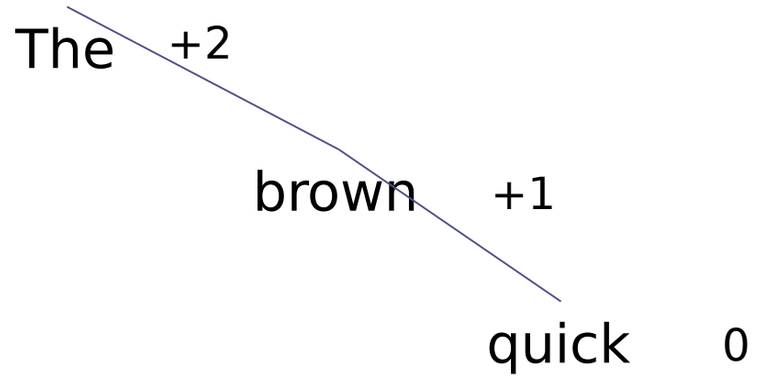
The overall tree is right-heavy (Right-Left)
parent balance = +2
right child balance = -1

The quick brown…



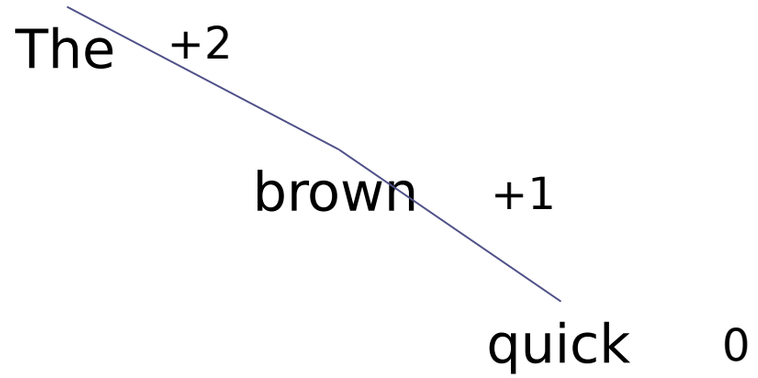
1. **Rotate right around the child**

The quick brown...



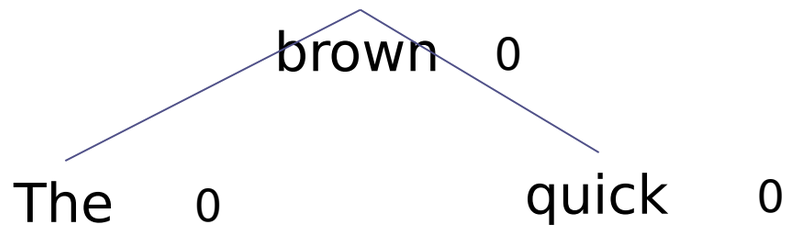
1. **Rotate right around the child**

The quick brown...



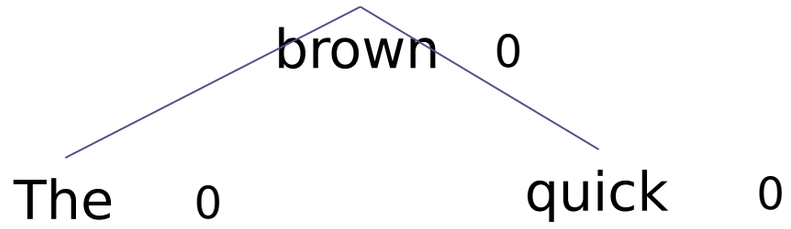
1. **Rotate right around the child**
2. **Rotate left around the parent**

The quick brown...



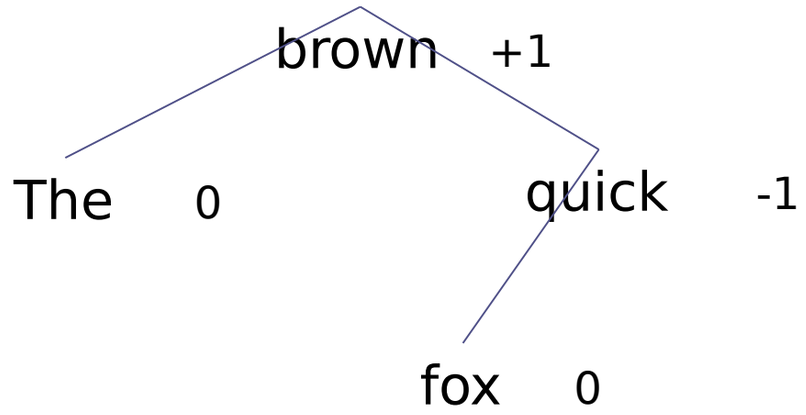
1. **Rotate right around the child**
2. **Rotate left around the parent**

The quick brown fox…



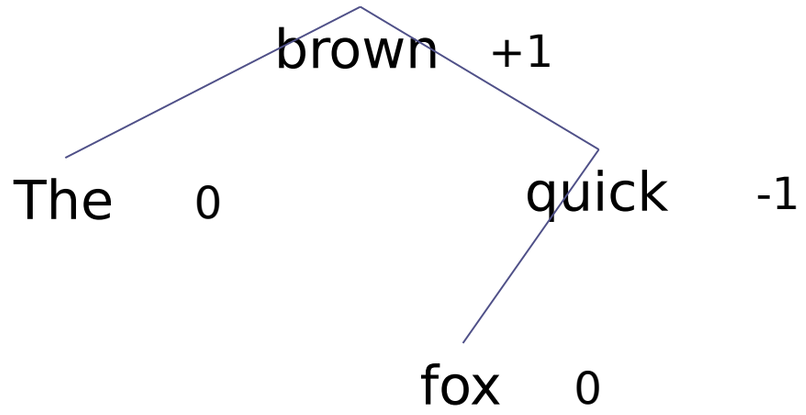
Insert *fox*

The quick brown fox…



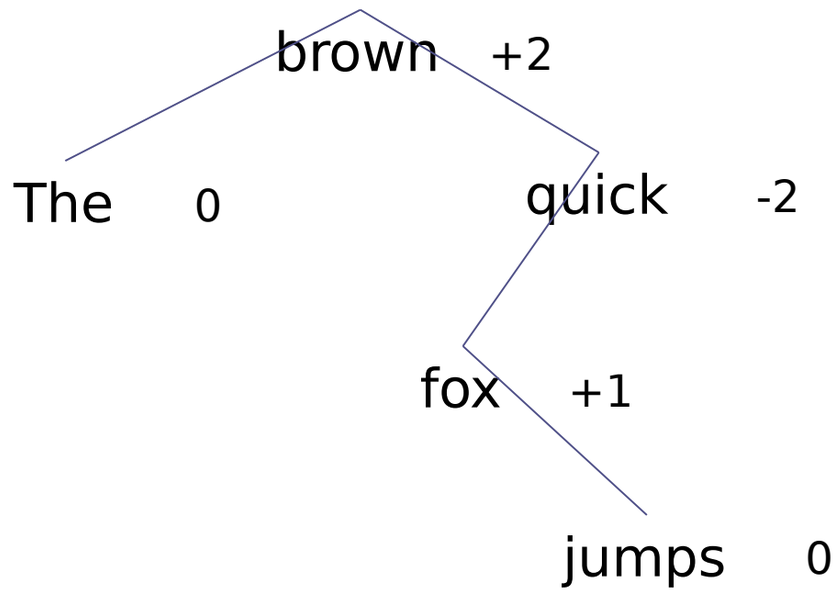
Insert *fox*

The quick brown fox jumps...



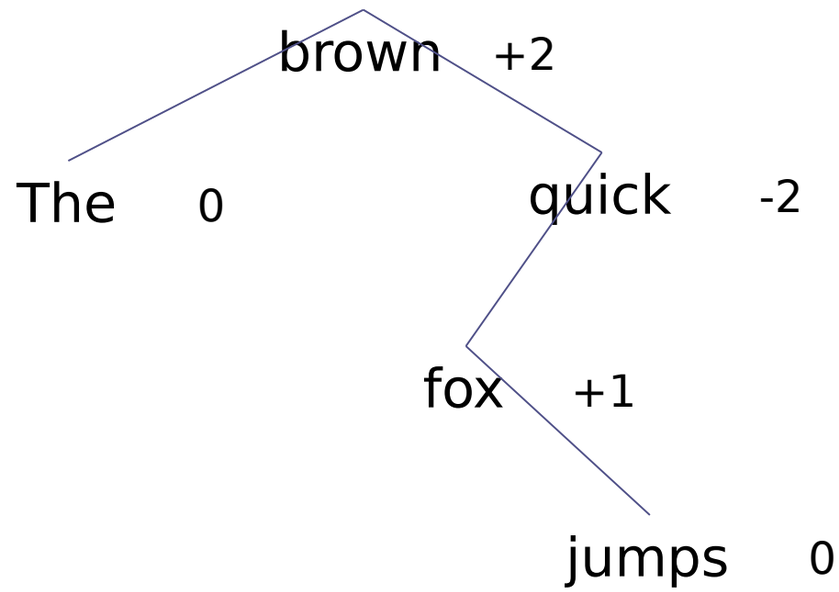
Insert *jumps*

The quick brown fox jumps...



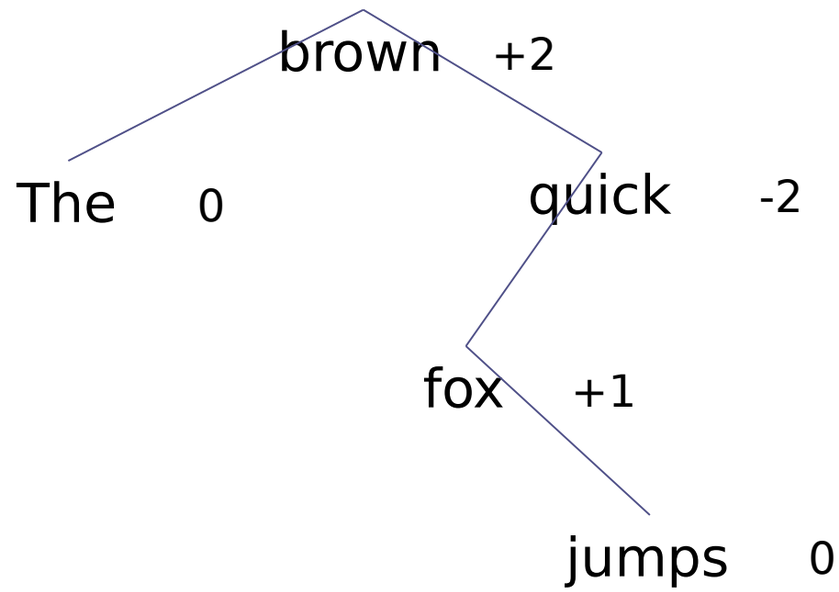
Insert *jumps*

The quick brown fox jumps...



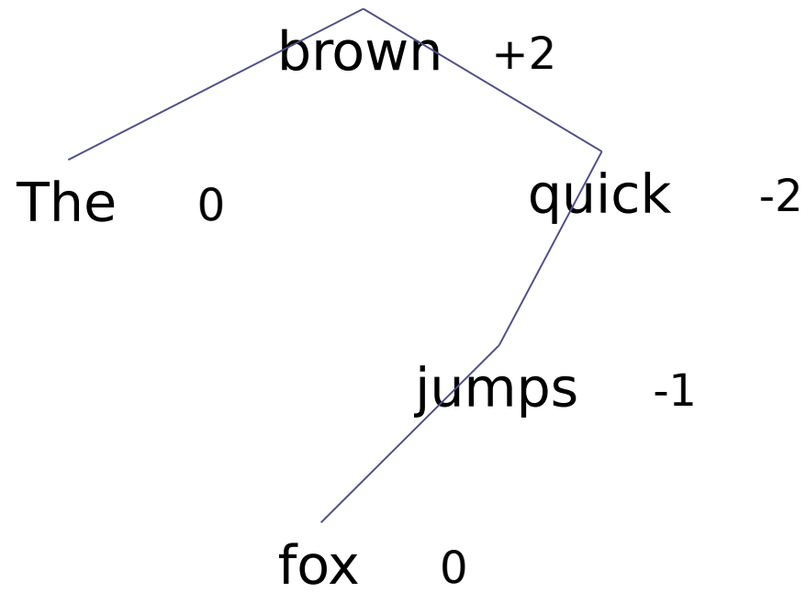
The tree is now left-heavy about *quick* (Left-Right case)

The quick brown fox jumps...



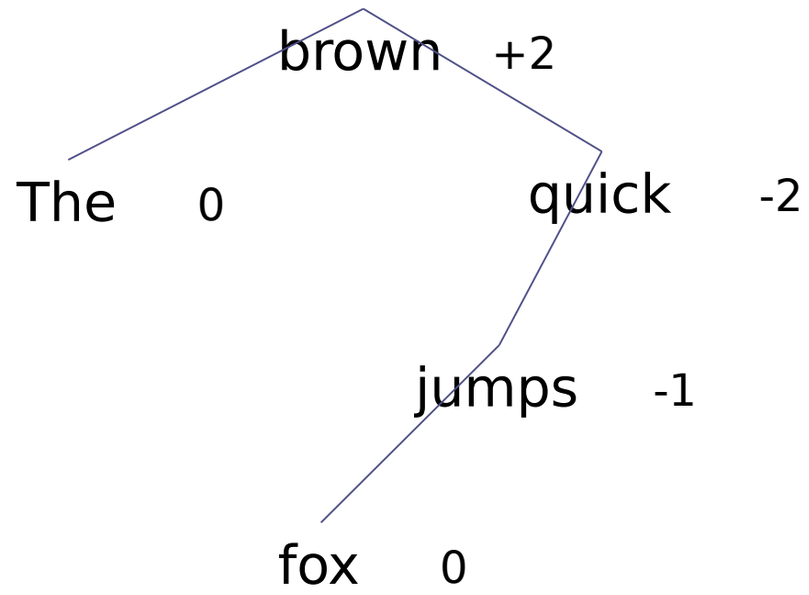
1.Rotate left around the child

The quick brown fox jumps...



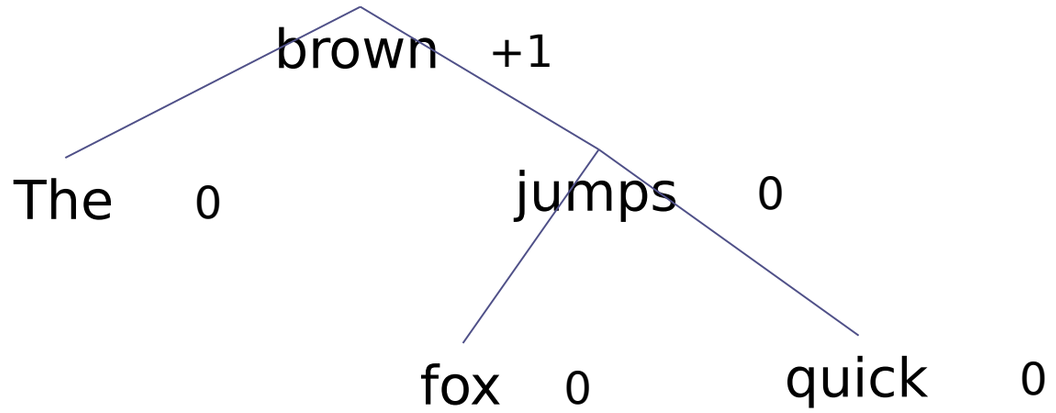
1.Rotate left around the child

The quick brown fox jumps...



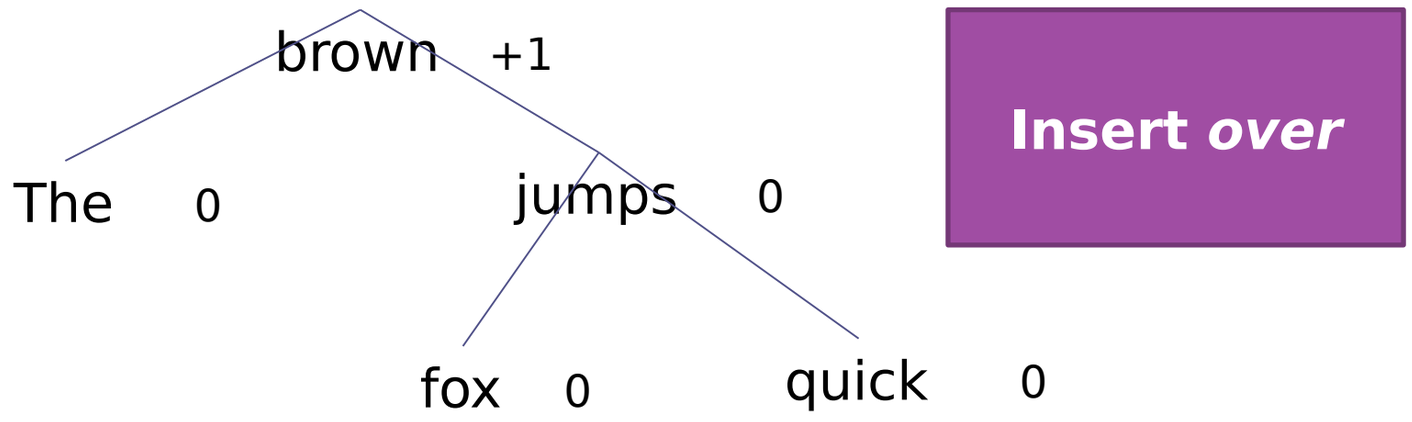
1. **Rotate left around the child**
2. **Rotate right around the parent**

The quick brown fox jumps...

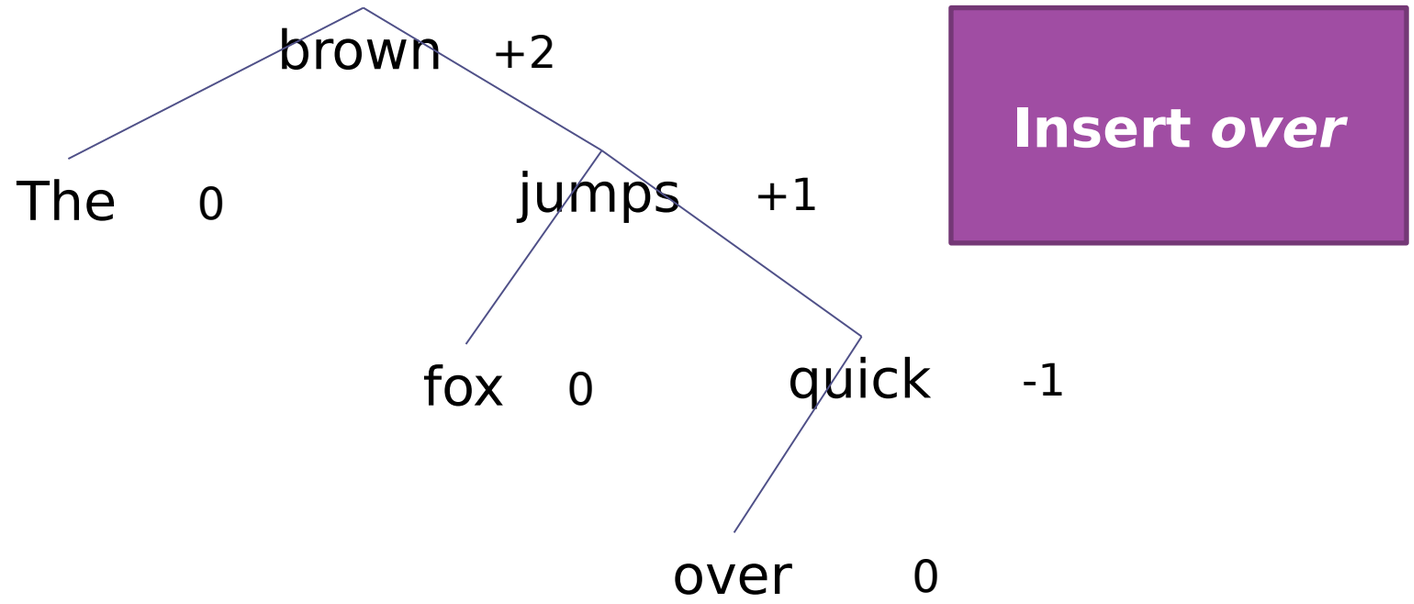


1. **Rotate left around the child**
2. **Rotate right around the parent**

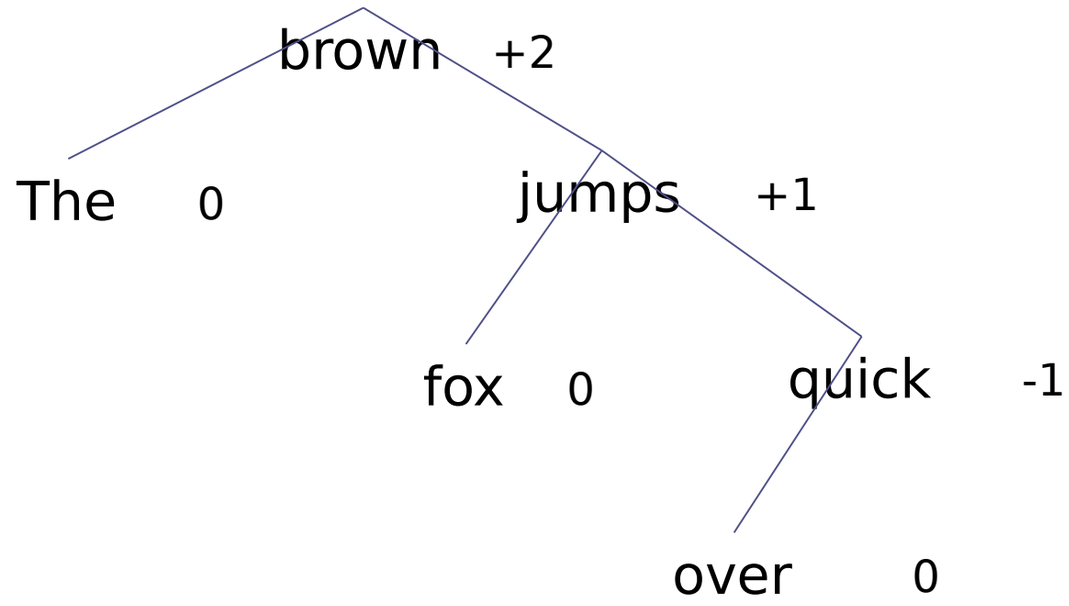
The quick brown fox jumps over...



The quick brown fox jumps over...

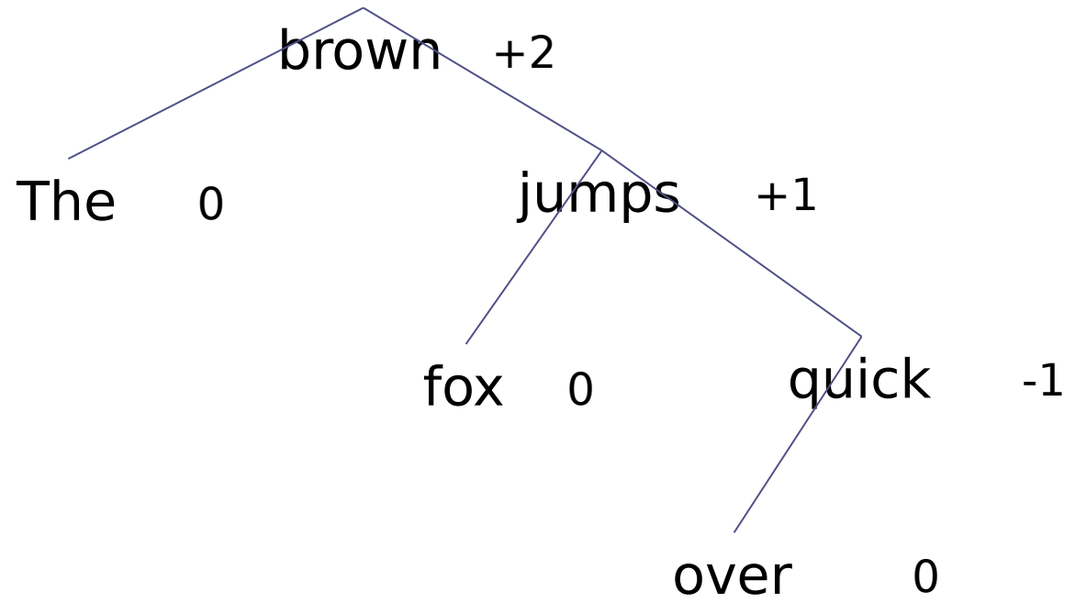


The quick brown fox jumps over...



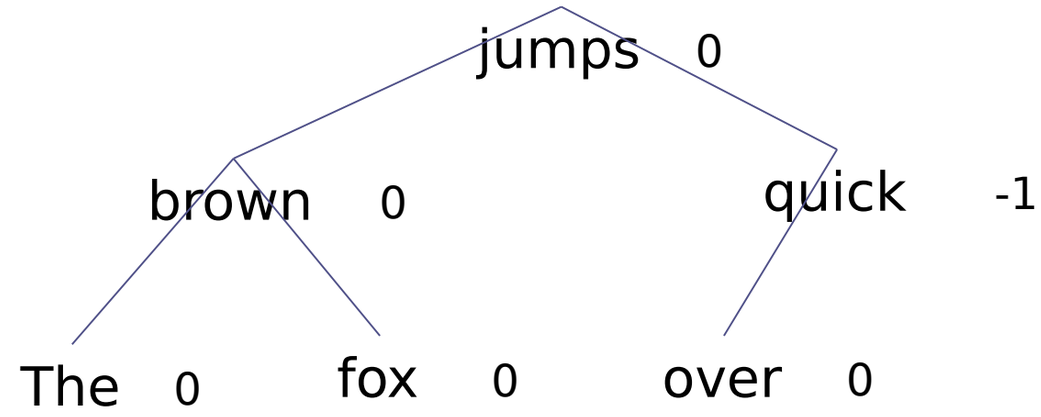
We now have a Right-Right imbalance

The quick brown fox jumps over...



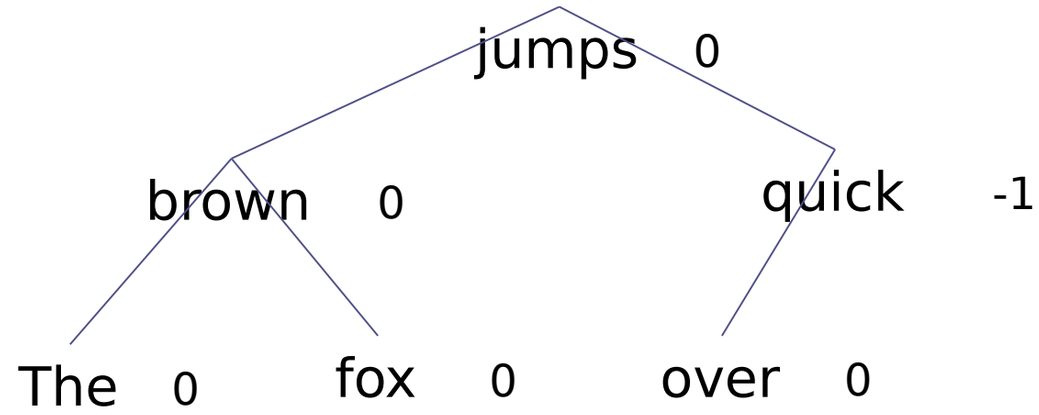
1. Rotate left around the parent

The quick brown fox jumps over...



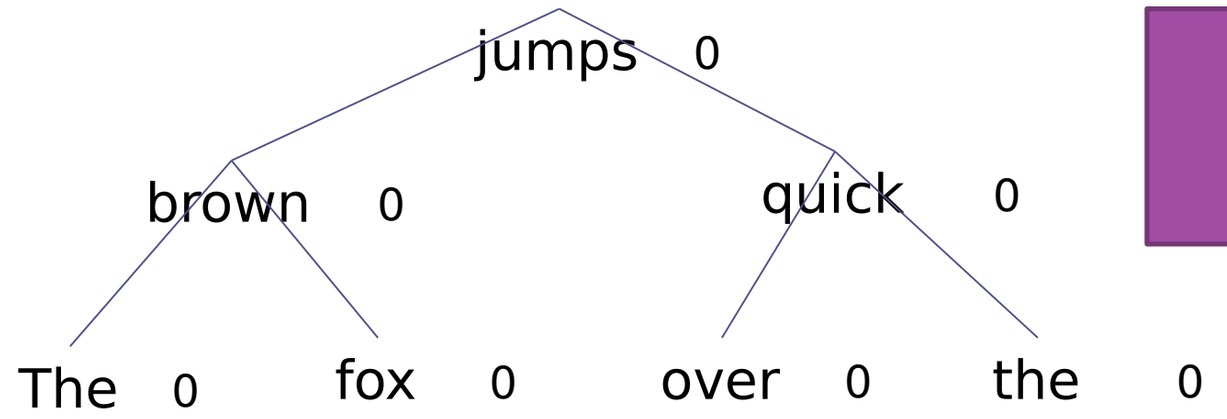
1. Rotate left around the parent

quick brown fox jumps over the...



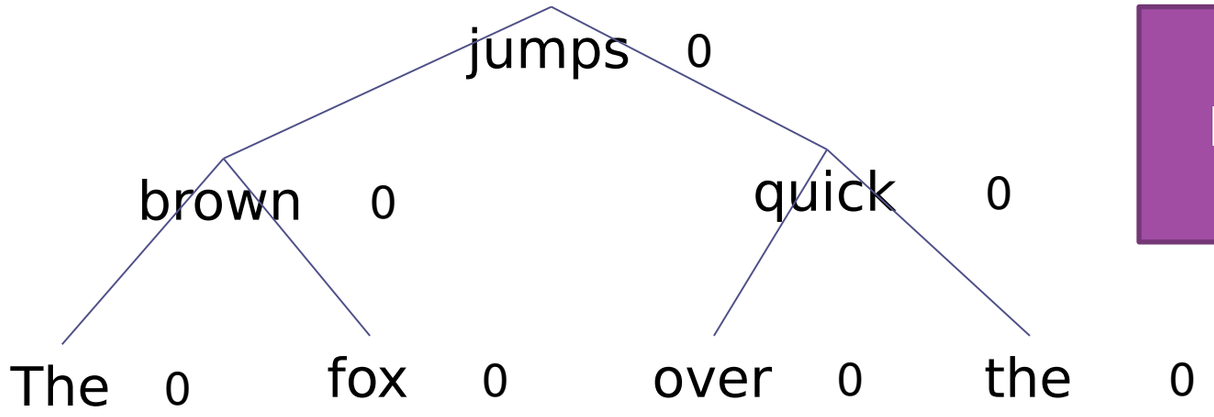
Insert *the*

The quick brown fox jumps over the...



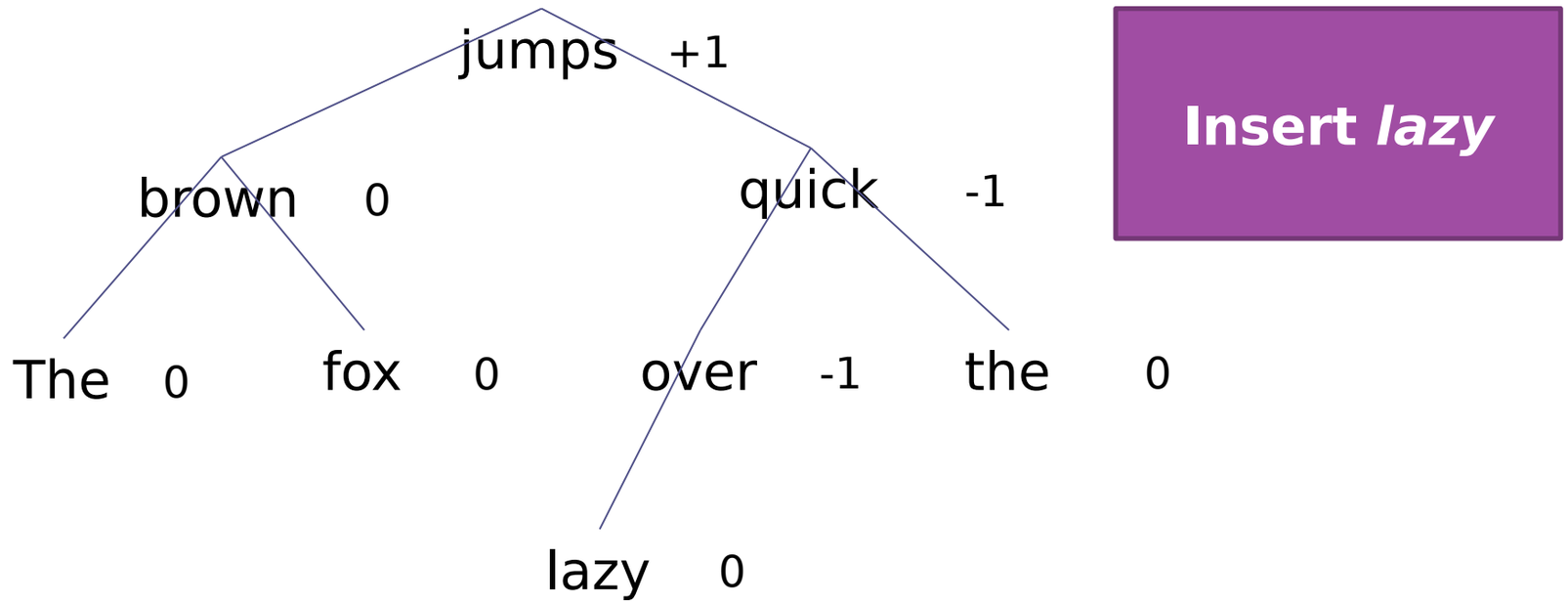
Insert *the*

quick brown fox jumps over the lazy...



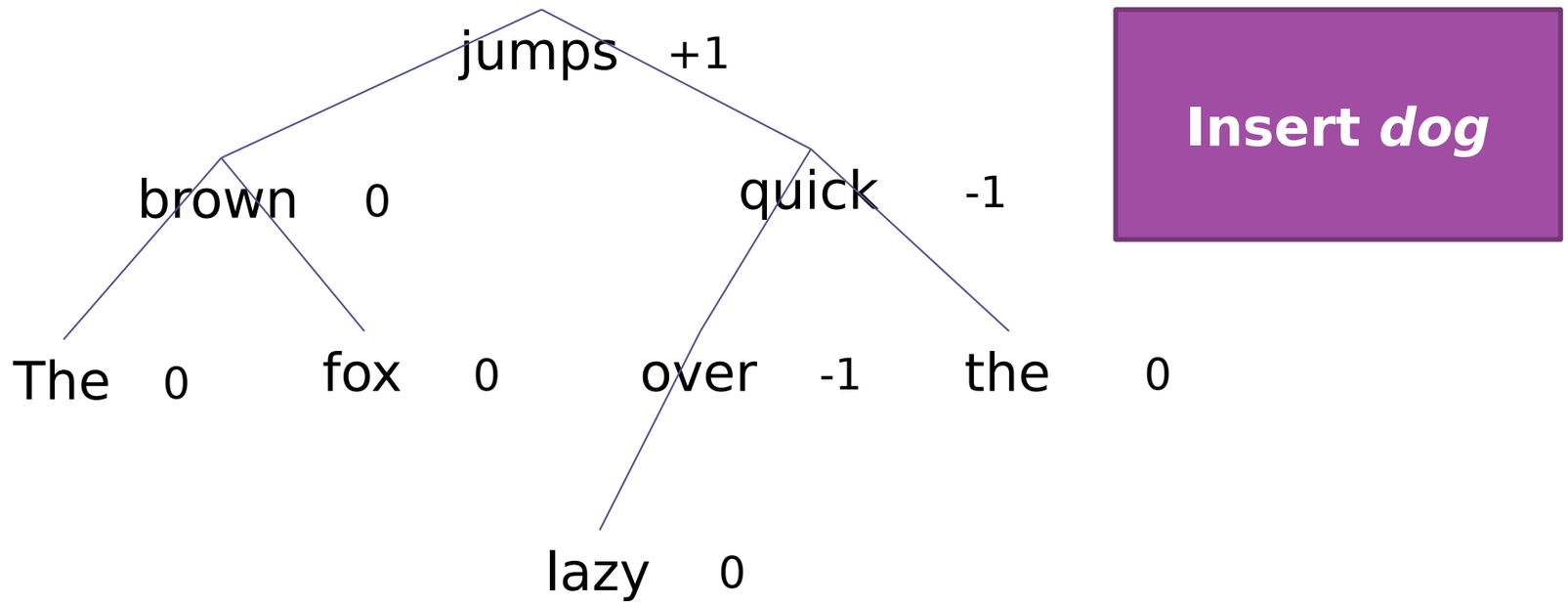
Insert *lazy*

quick brown fox jumps over the lazy...

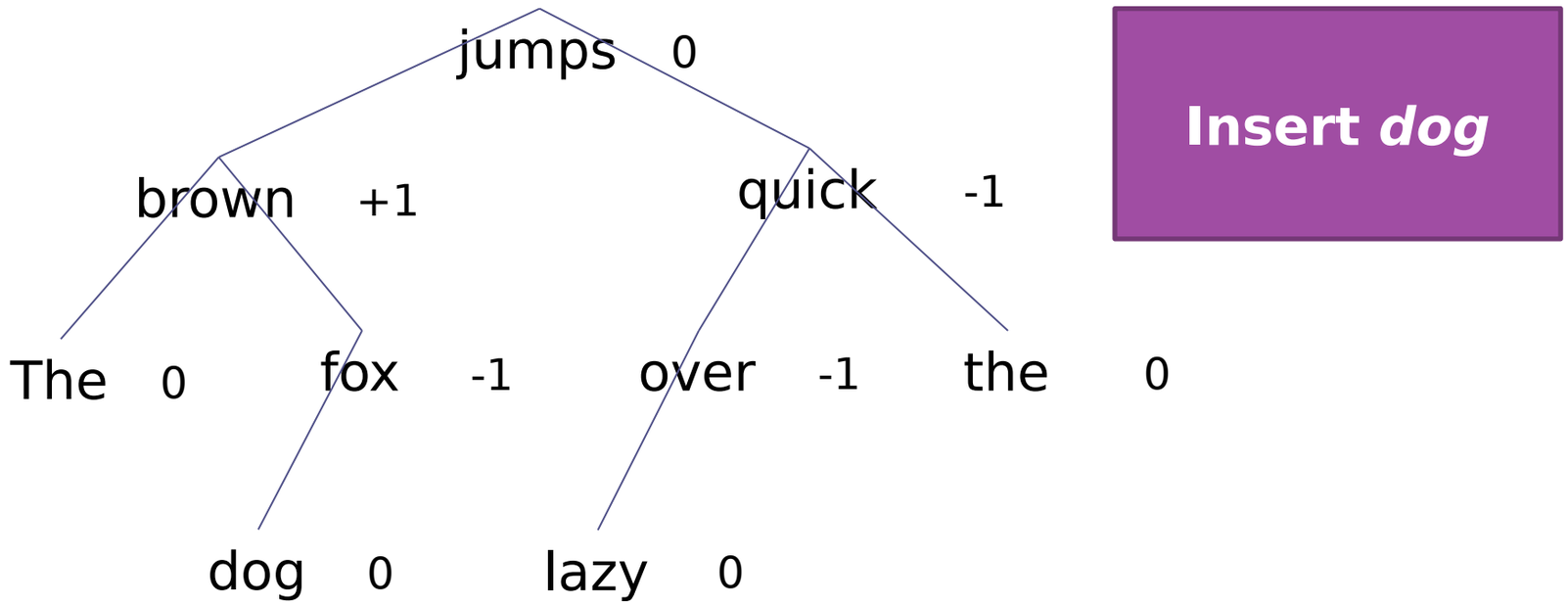


Insert *lazy*

quick brown fox jumps over the lazy dog



quick brown fox jumps over the lazy dog!



Insert *dog*

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

AVL trees

A balanced BST that maintains balance by *rotating* the tree

- Insertion: insert as in a BST and move upwards from the inserted node, rotating unbalanced nodes
- Deletion: delete as in a BST and move upwards from the node that disappeared, rotating unbalanced nodes

Worst-case 1.44log n, typical log n comparisons for any operation – very balanced. This means lookups are quick.