Search trees

AVL tree

Operations Find, Insert, Delete

Rotations L, R, LR, RL

B-tree

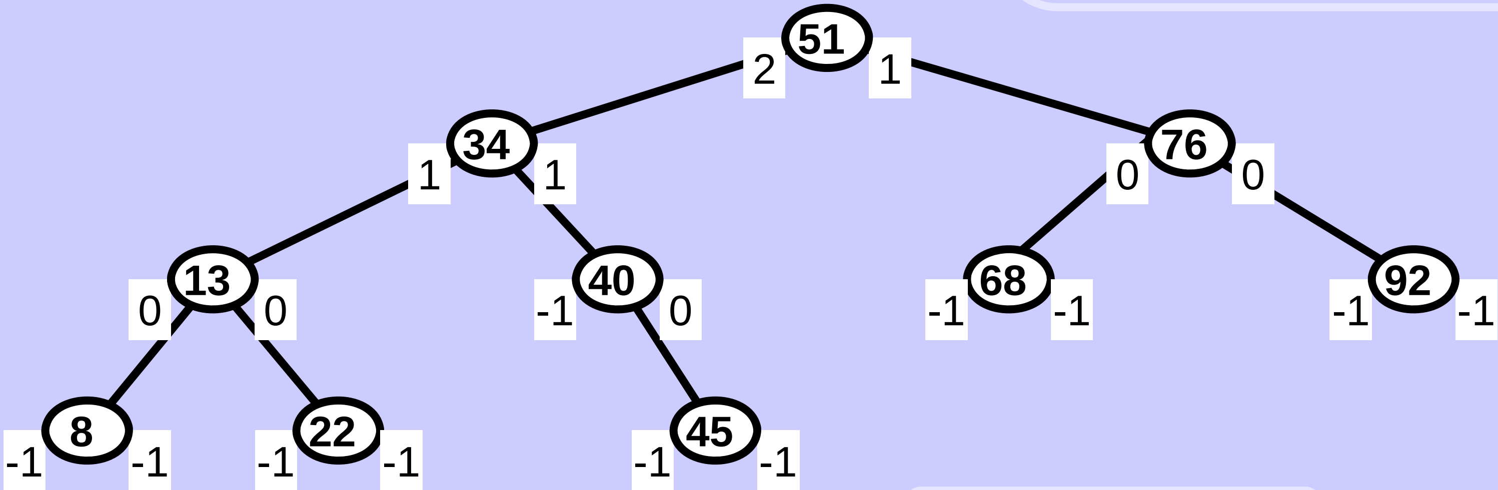
Operations Find, Insert, Delete

Single phase and multi phase update strategies

AVL tree -- G.M. Adelson-Velskij & E.M. Landis, 1962

AVL tree is a BST with additional properties which keep it acceptably balanced.

Operations
Find, Insert, Delete
also apply to AVL tree.



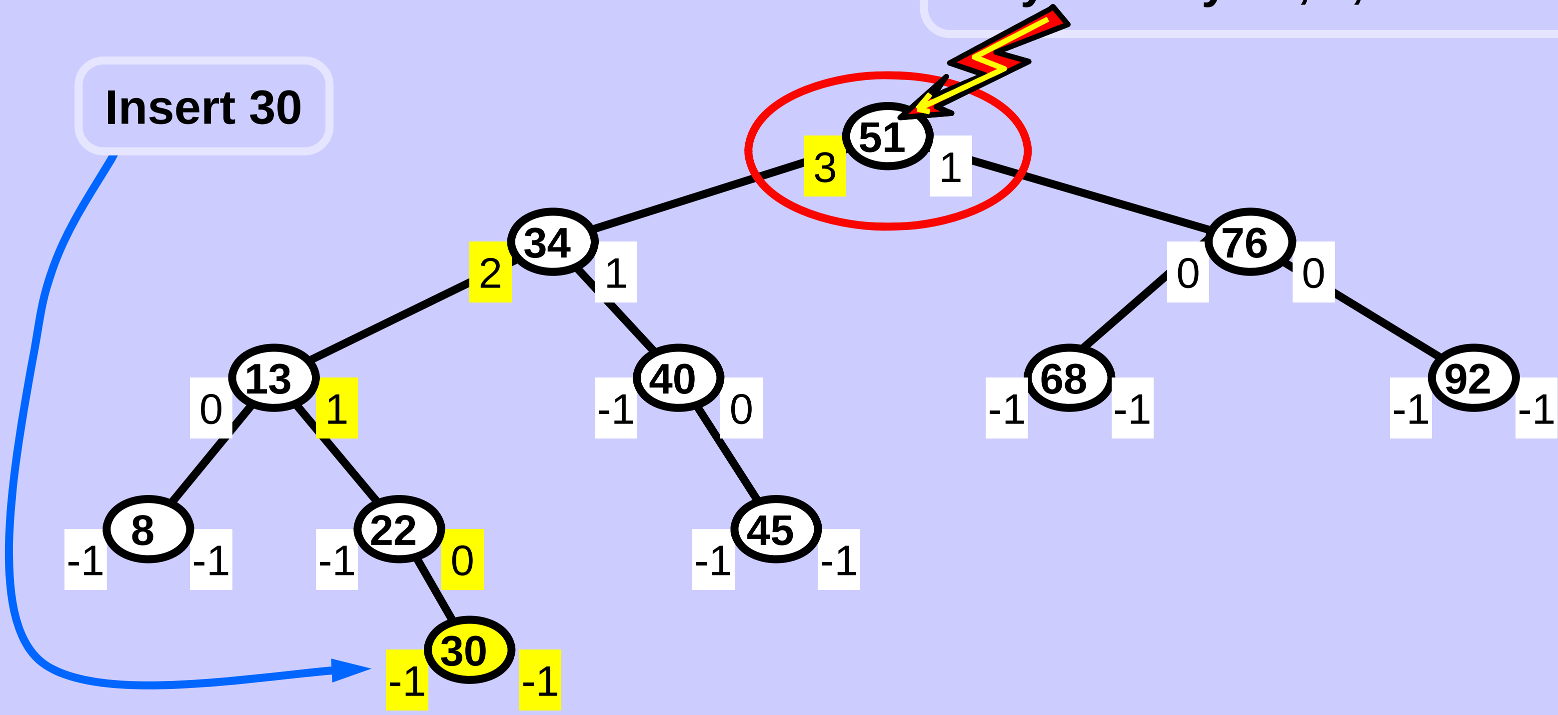
There are two integers associated with each node X:
Depth of the left subtree of X and depth of the right subtree of X.
Note: Depth of an empty tree is -1.

AVL rule:

The difference of the heights of the left and the right subtree may be only -1 or 0 or 1 in each node of the tree.

Inserting a node may disbalance the AVL tree

The subtrees height difference may be only -1, 0, 1.

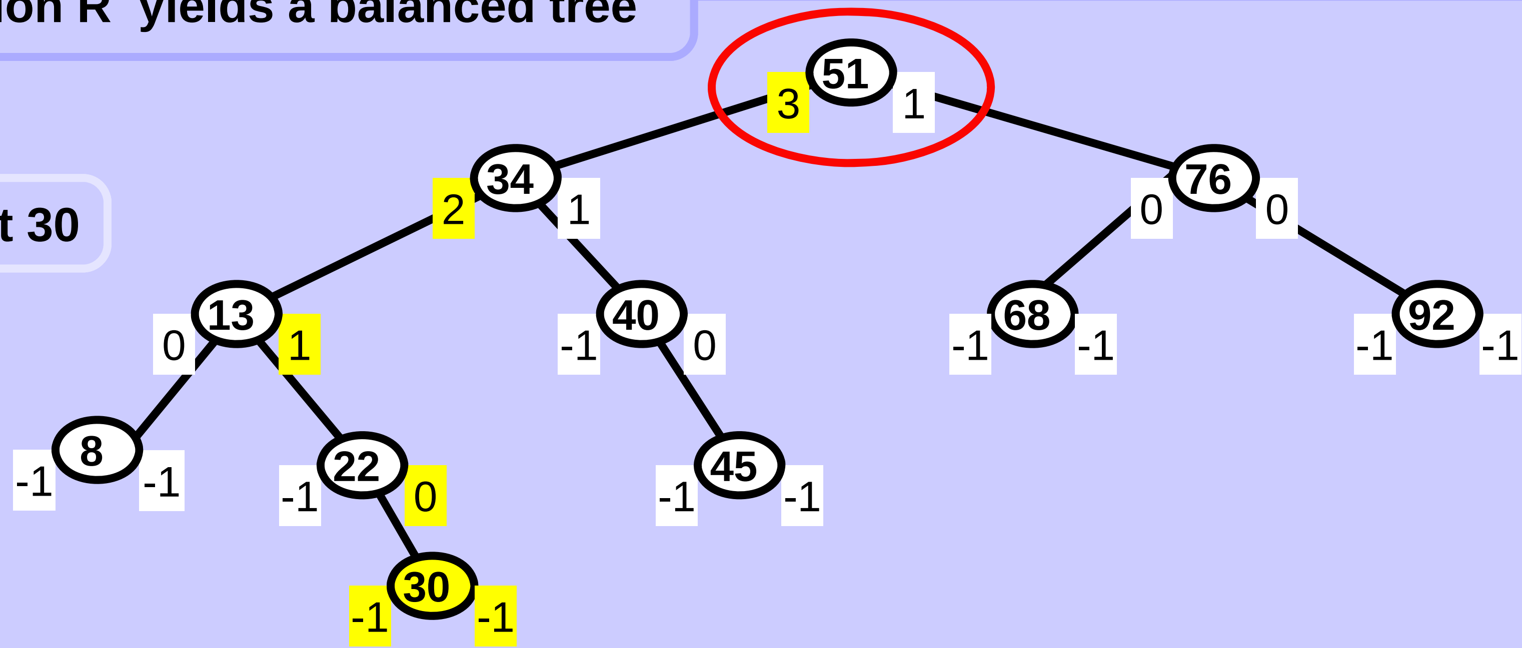


Changed depths

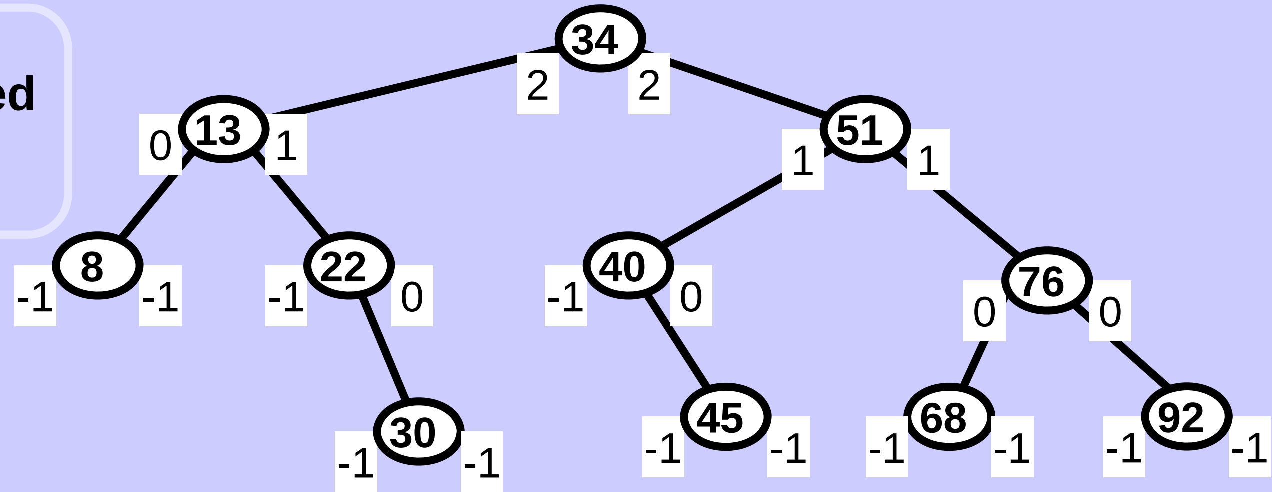
Left subtree of node 51 is too deep, the tree is no more an AVL tree.

Rotation R yields a balanced tree

Insert 30

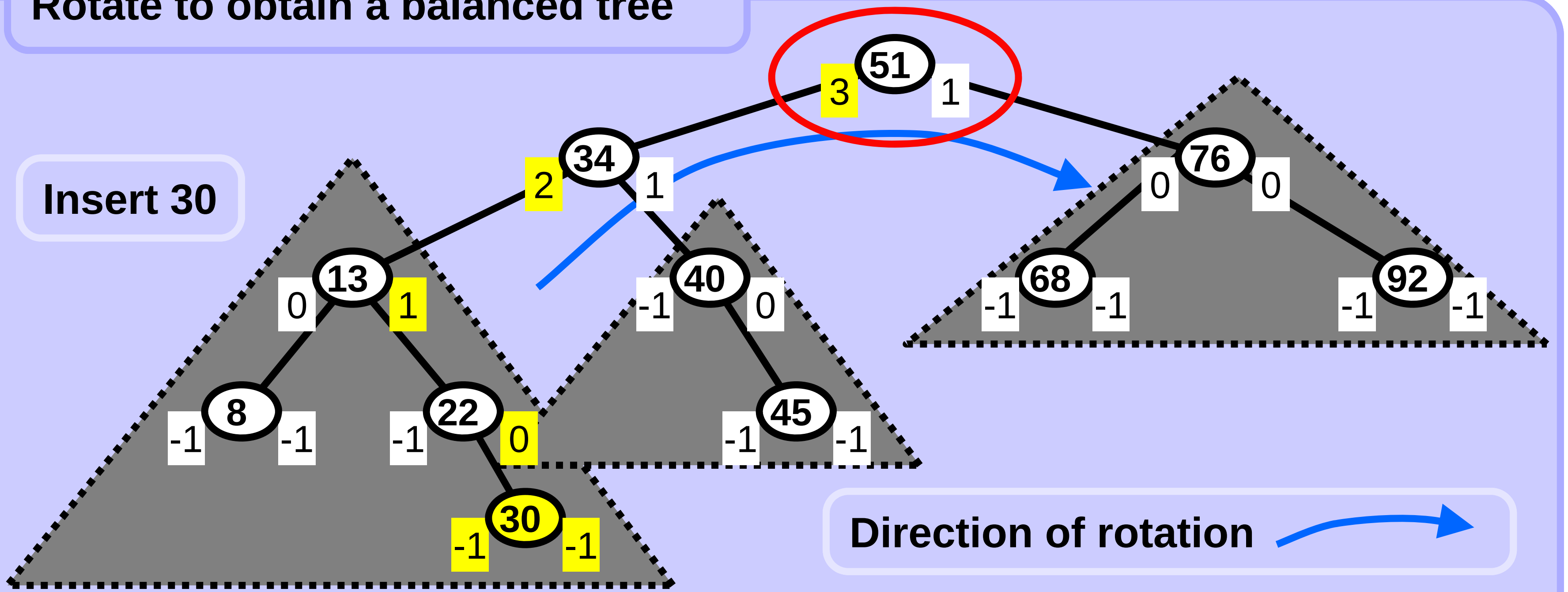


The tree was balanced by single R rotation



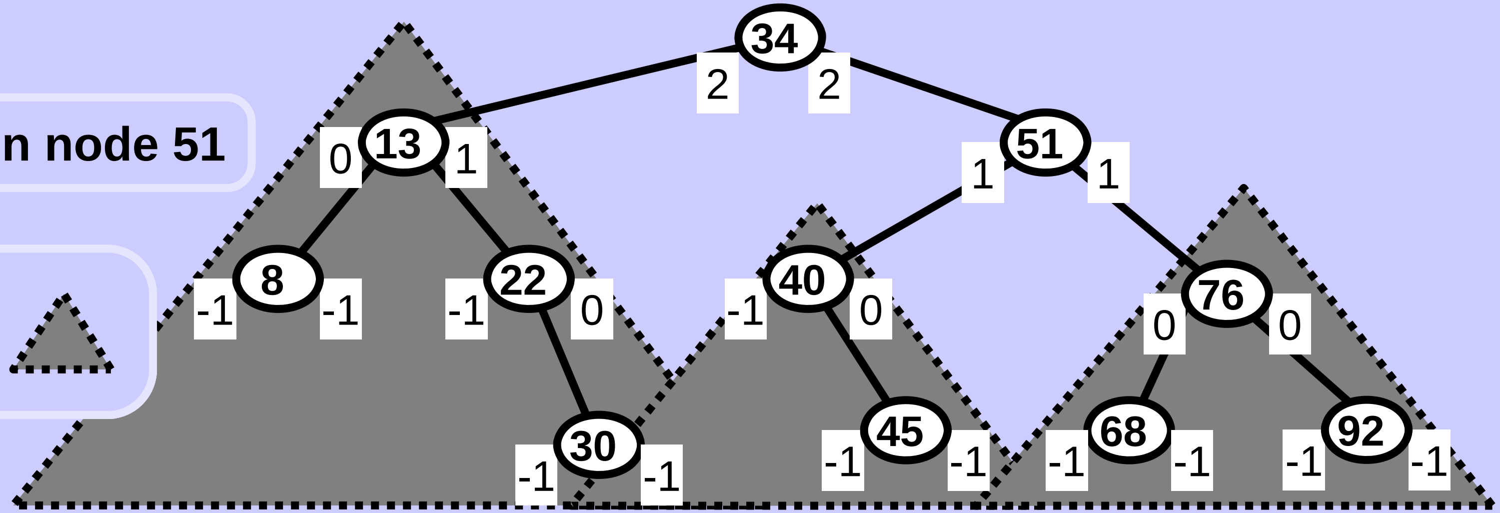
Rotate to obtain a balanced tree

Insert 30



Direction of rotation

Rotation R in node 51



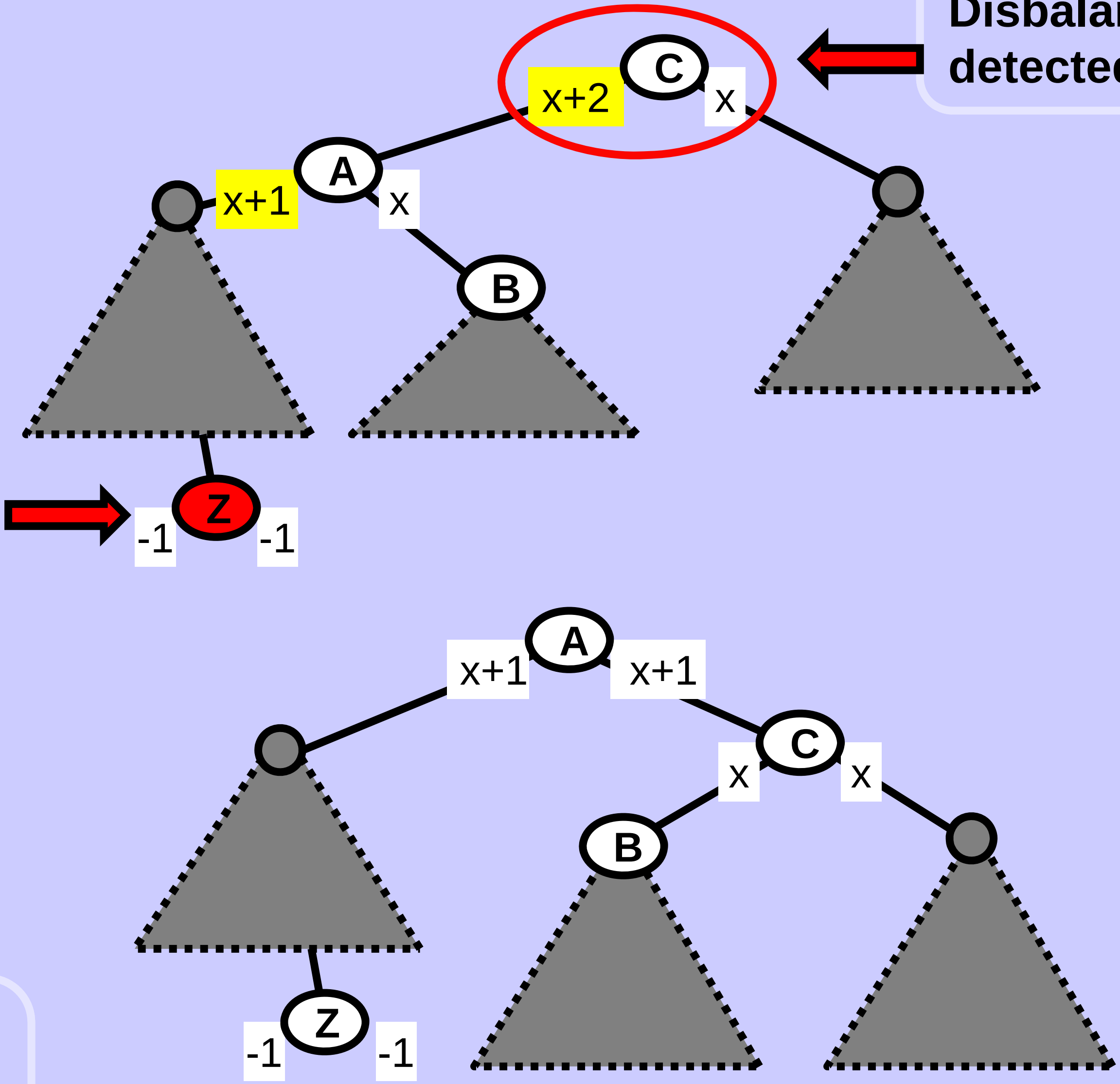
Unaffected subtrees

Rotation R in general

Disbalance detected

Before

Disbalancing node

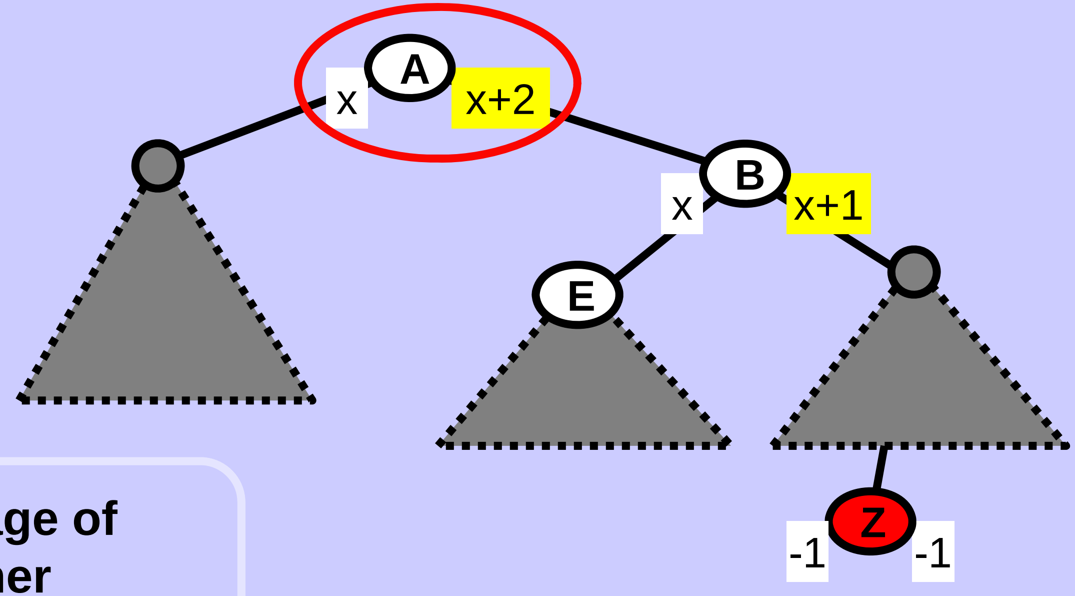


After

Unaffected subtrees

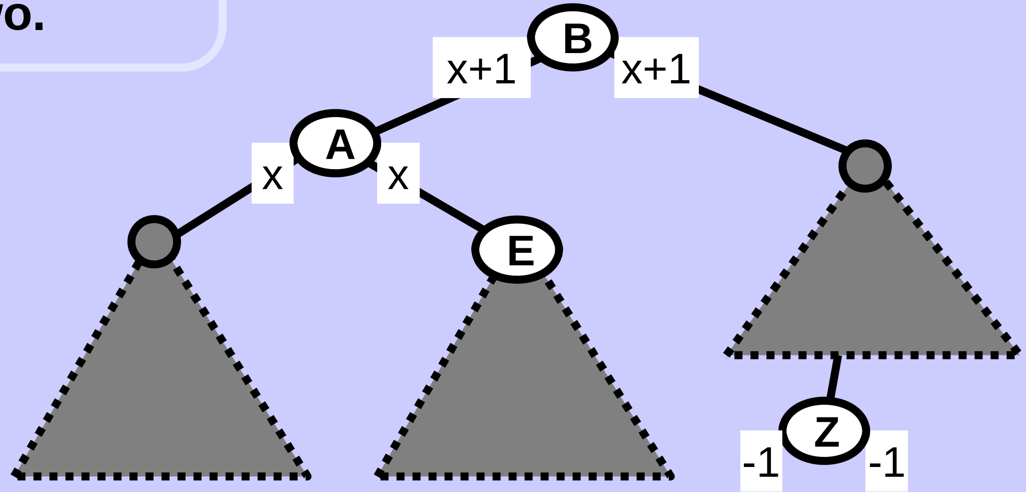
Rotation L in general, before and after

Before

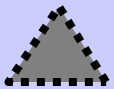


Rotation L is a mirror image of rotation R, there is no other difference between the two.

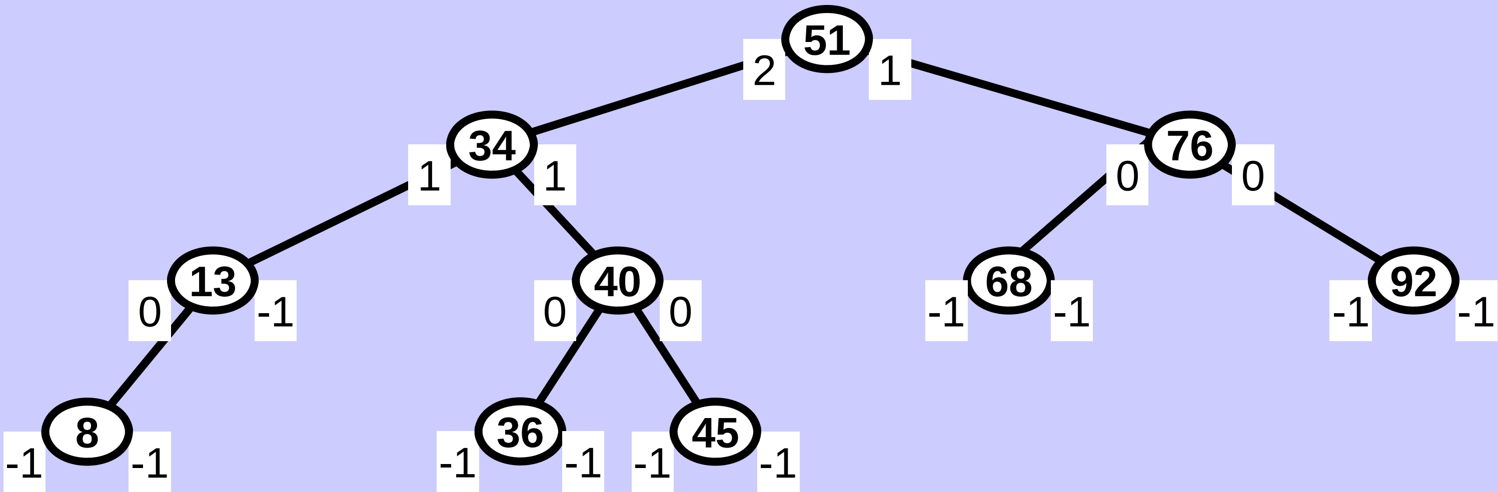
After



Unaffected subtrees



AVL tree

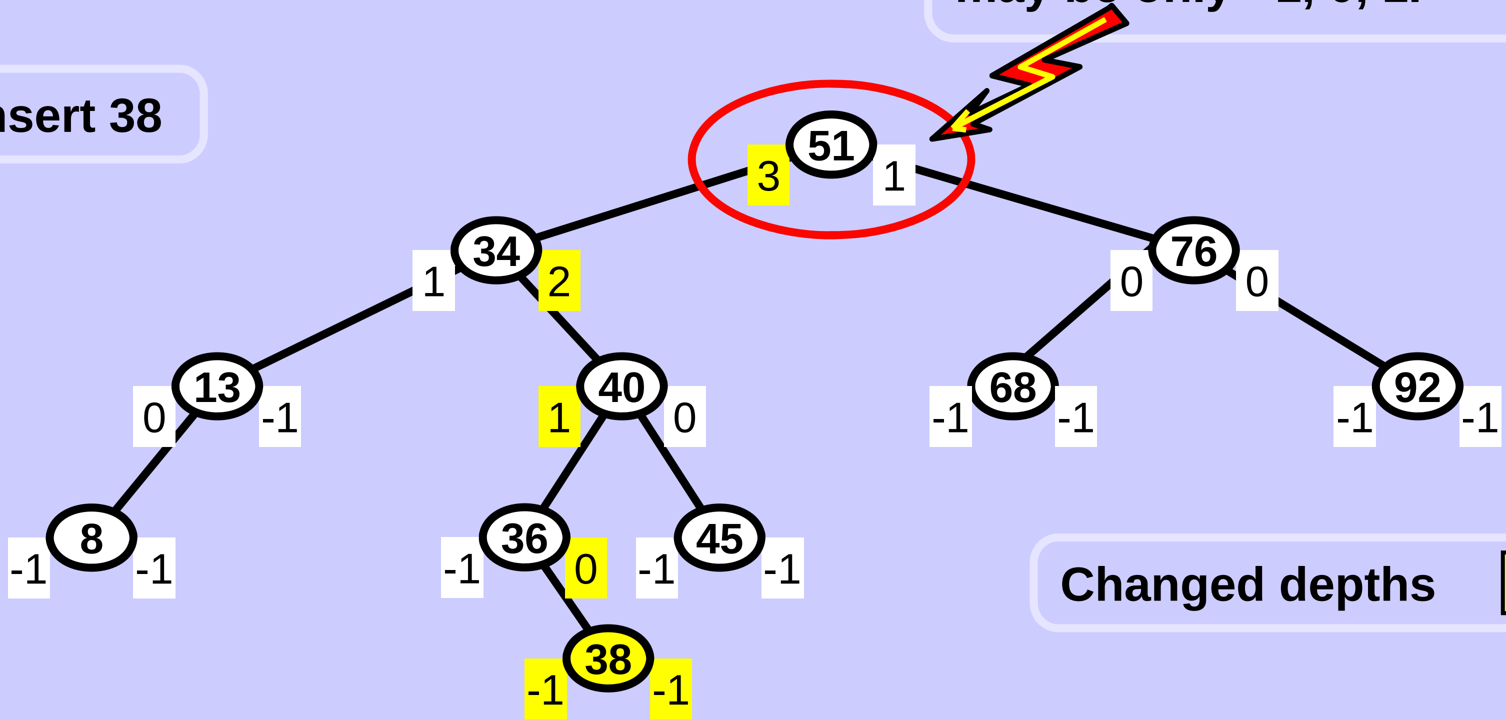


Demonstration AVL tree for rotation LR

Inserting a node may disbalance the AVL tree

The subtrees height difference may be only -1, 0, 1.

Insert 38



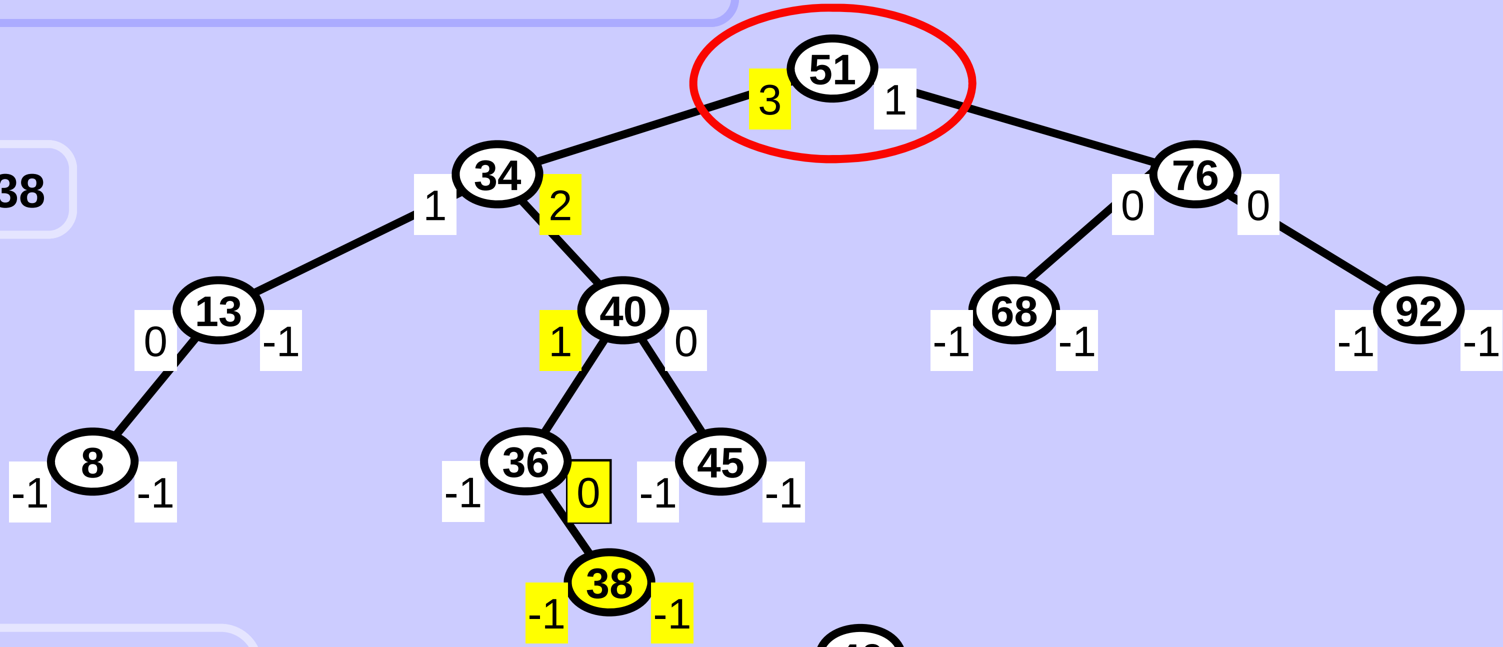
Changed depths

Left subtree of node 51 is too deep, the tree is no more an AVL tree.

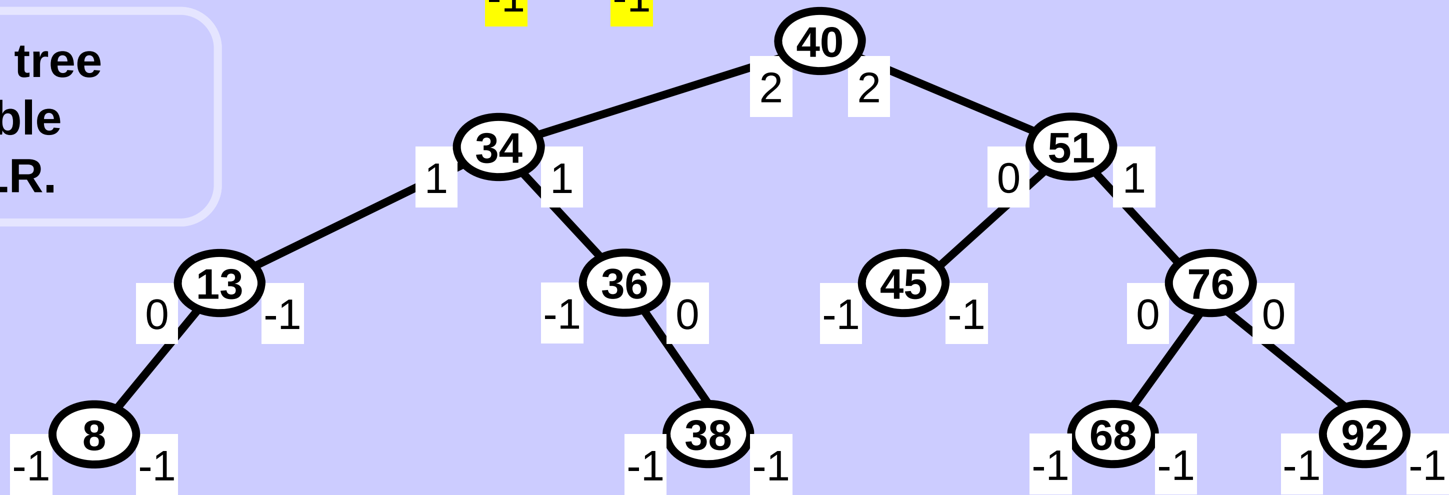
Rotation R would not help, the right subtree of node 34 would become relatively too deep compared to the new left subtree of the root.

Rotation LR yields a balanced tree

Insert 38

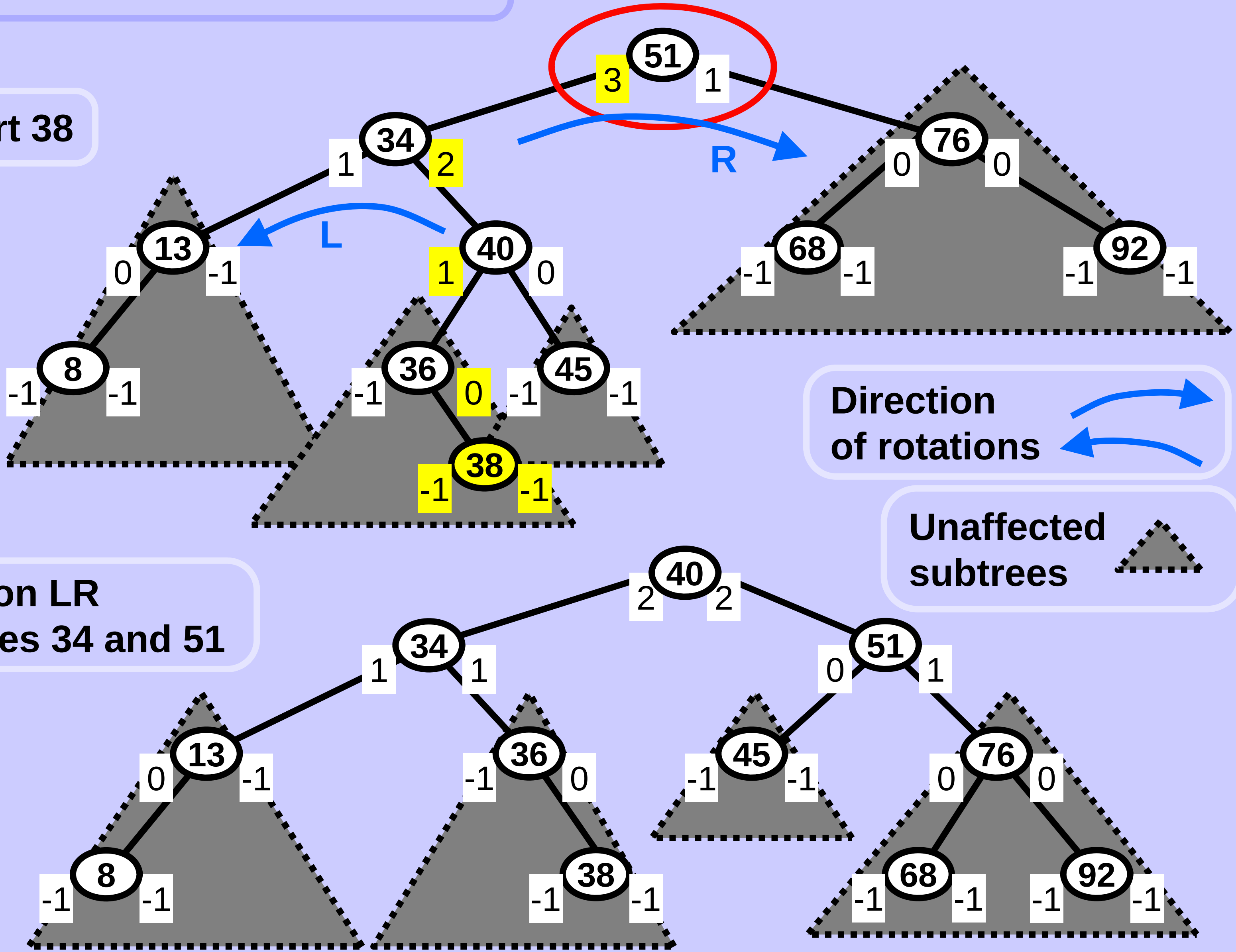


Balanced tree after double rotation LR.



Rotate to obtain a balanced tree

Insert 38

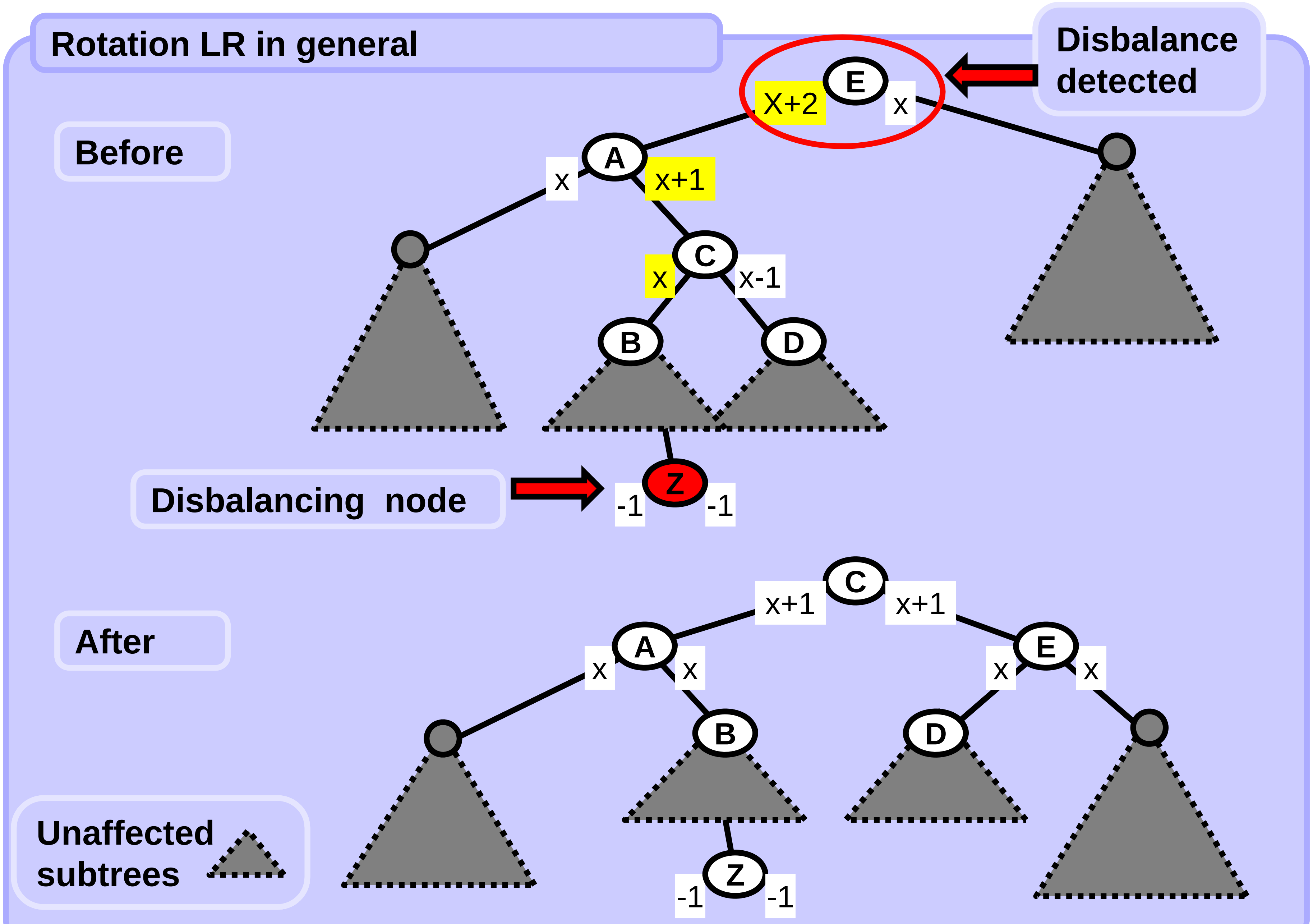


Direction of rotations

Unaffected subtrees

Rotation LR in nodes 34 and 51

Rotation LR in general



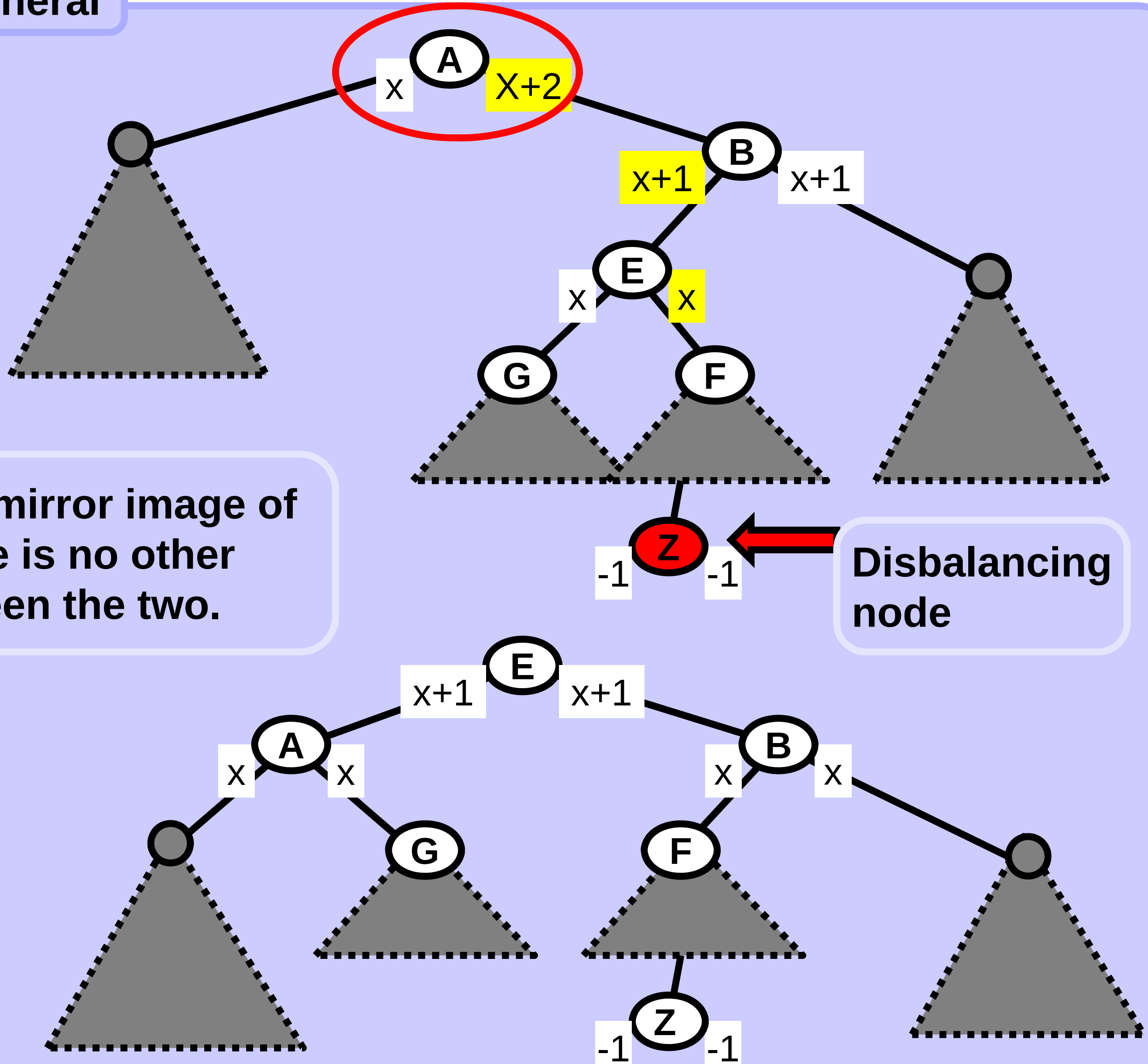
Rotation RL in general

Before

Rotation RL is a mirror image of rotation LR, there is no other difference between the two.

After

Unaffected subtrees



Rules for aplying rotations L, R, LR, RL in Insert operation

Travel from the inserted node up to the root and update subtree depths in each node along the path.

If a node is disbalanced and you came to it along two consecutive edges

- **both in the up and *right* direction perform rotation R in this node,**
- **both in the up and *left* direction perform rotation L in this node,**
- **first in the in the up and *left* and then in the up and *right* direction perform rotation LR in this node,**
- **first in the in the up and *right* and then in the up and *left* direction perform rotation RL in this node,**

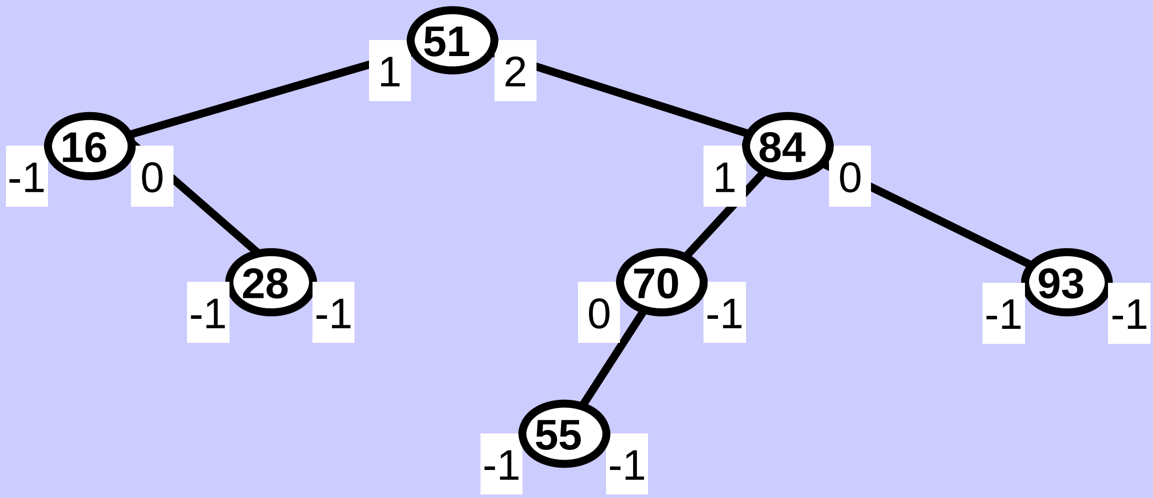
One rotation in the Insert operation balances correctly the AVL tree.

After one rotation in the Delete operation the AVL tree might still not be balanced, all nodes on the path to the root have to be checked.

Delete in AVL tree

Demonstration AVL tree for rotation after deletion

Delete 16

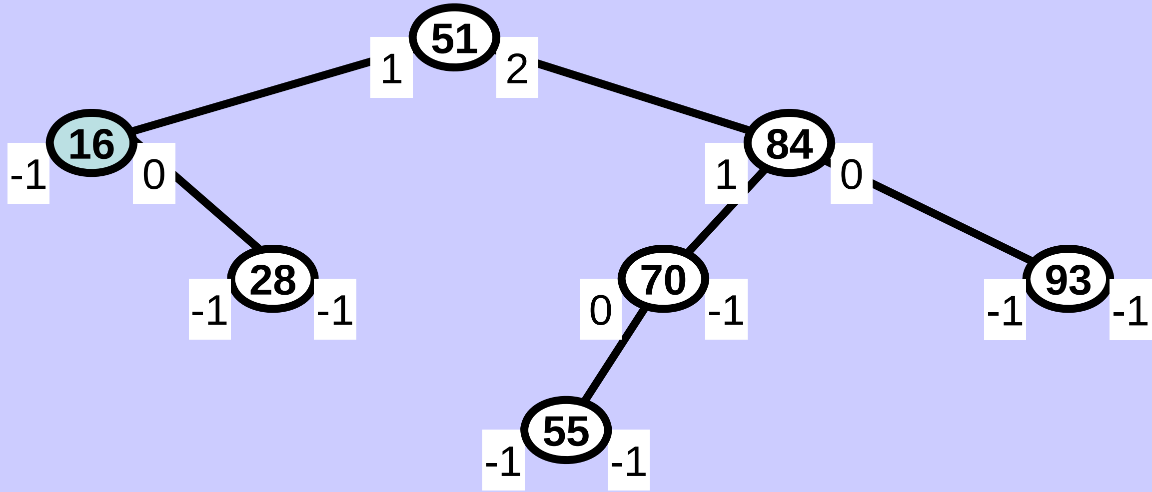


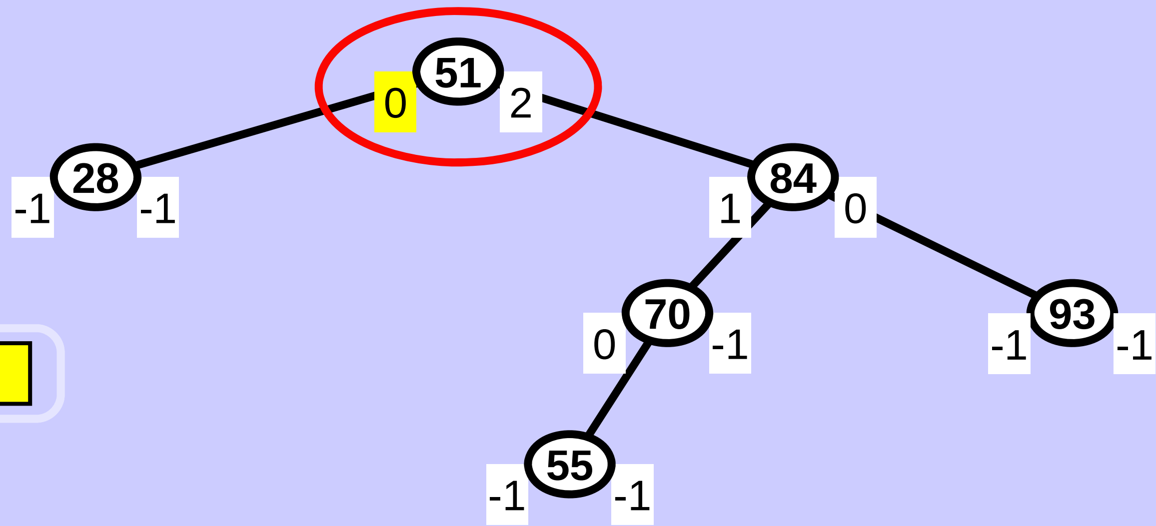
1. Remove the node using the same method as in BST.

2. Travel from the place of deletion up to the root. Update subtree heights in each node, and if necessary apply the corresponding rotation.

Delete in AVL tree

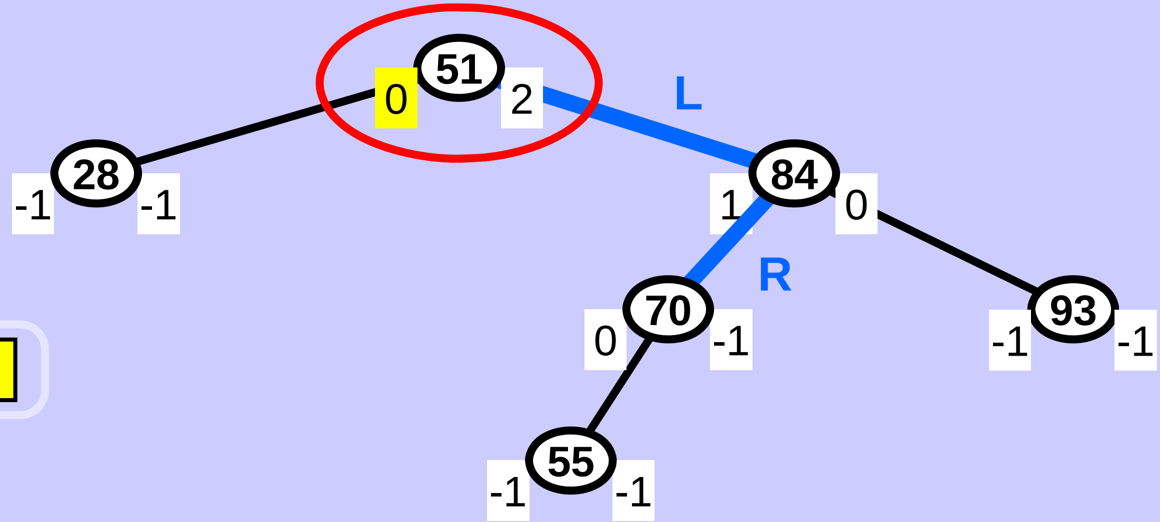
Delete 16





Changed depths

Delete in AVL tree

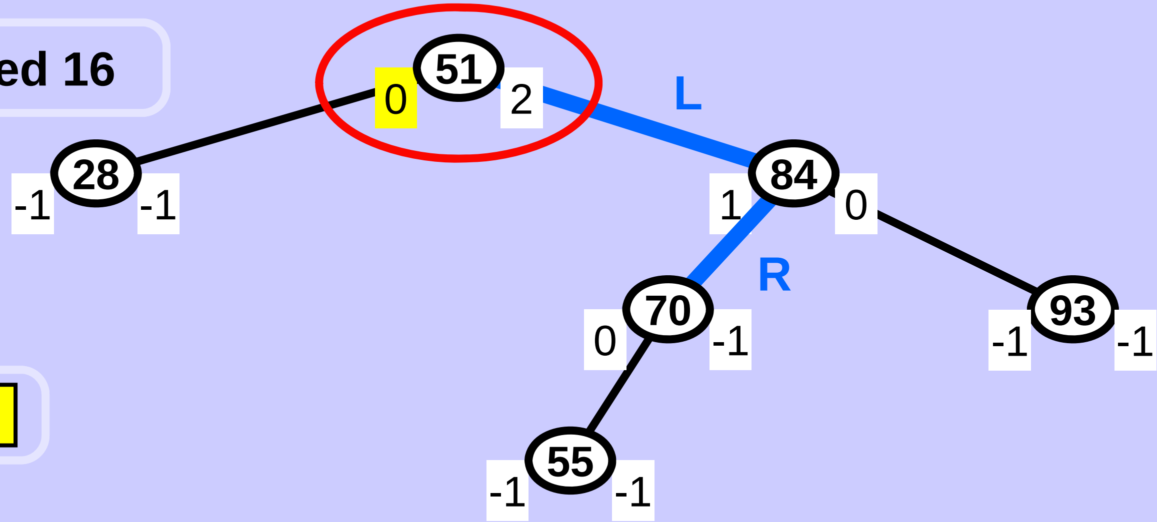


Changed depths

In the disbalanced node (51), check the root (84) of the subtree opposite to the one which you came from.
If the heights of subtrees of that root are equal apply a single rotation R or L.
If the hight of the more distant subtree of that root is bigger than the height of the less distant one apply a single rotation R or L.
In the remaining case apply a double rotation RL or LR.
In this example, apply RL.

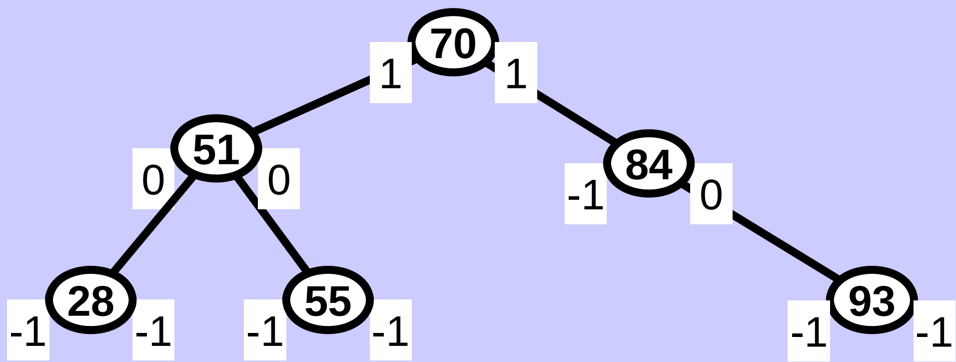
Delete in AVL tree

Deleted 16



Changed depths

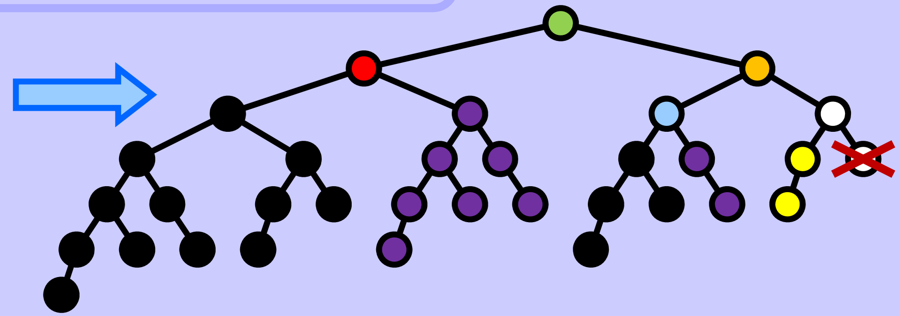
After rotation RL
in nodes 84 and 51



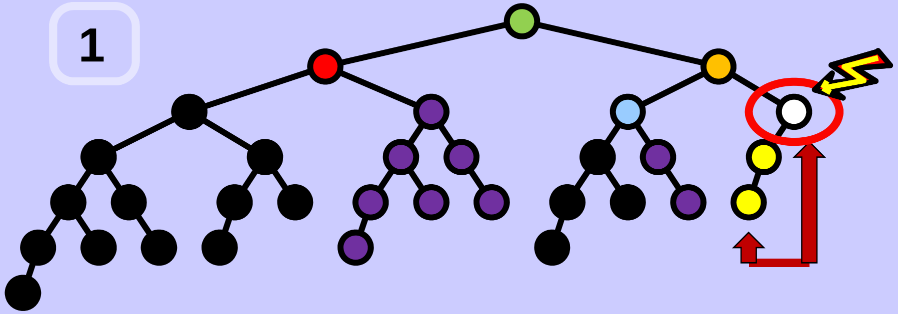
Possibility of multiple rotations in operation Delete.

Example.
The AVL tree is originally balanced.

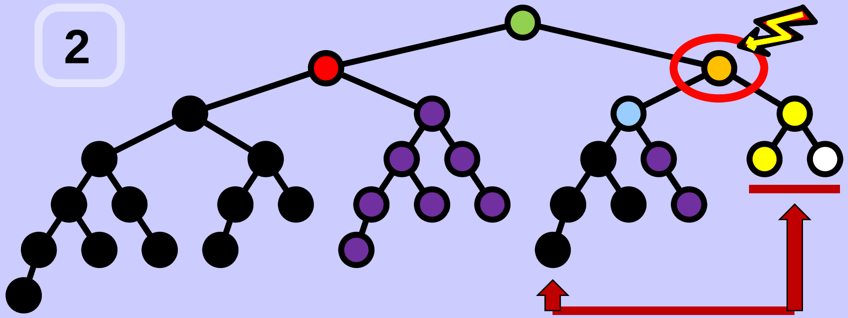
Delete the rightmost key.



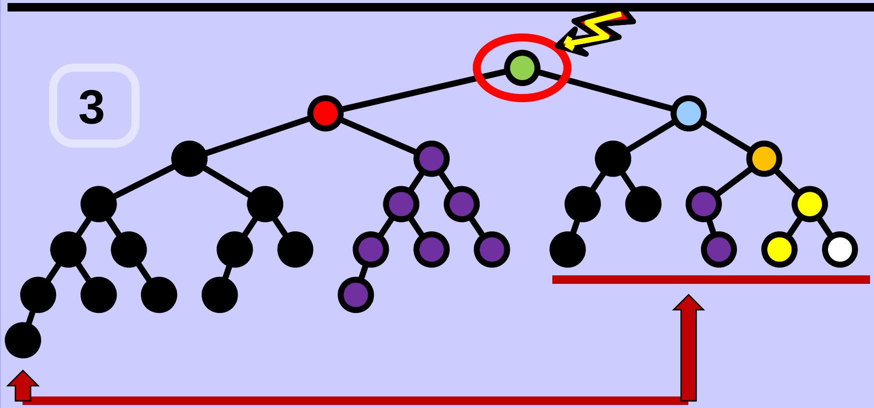
1



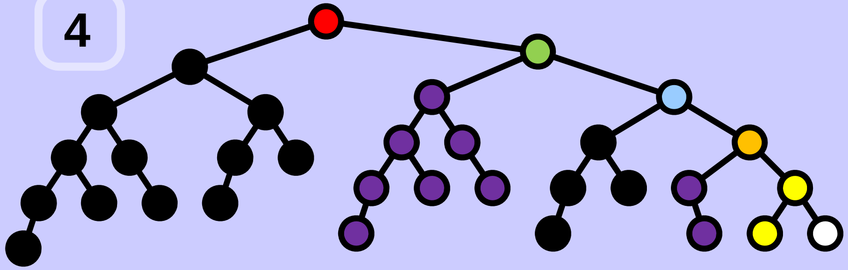
2



3



4



Balanced.

Asymptotic complexities of Find, Insert, Delete in BST and AVL

	BST with n nodes		AVL tree with n nodes
Operation	balanced (rarely)	not balanced (typically)	always balanced
Find	$O(\log(n))$	$O(n)$	$O(\log(n))$
Insert	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$
Delete	$O(\log(n))$	$O(n)$	$\Theta(\log(n))$

B-tree -- Rudolf Bayer, Edward M. McCreight, 1972

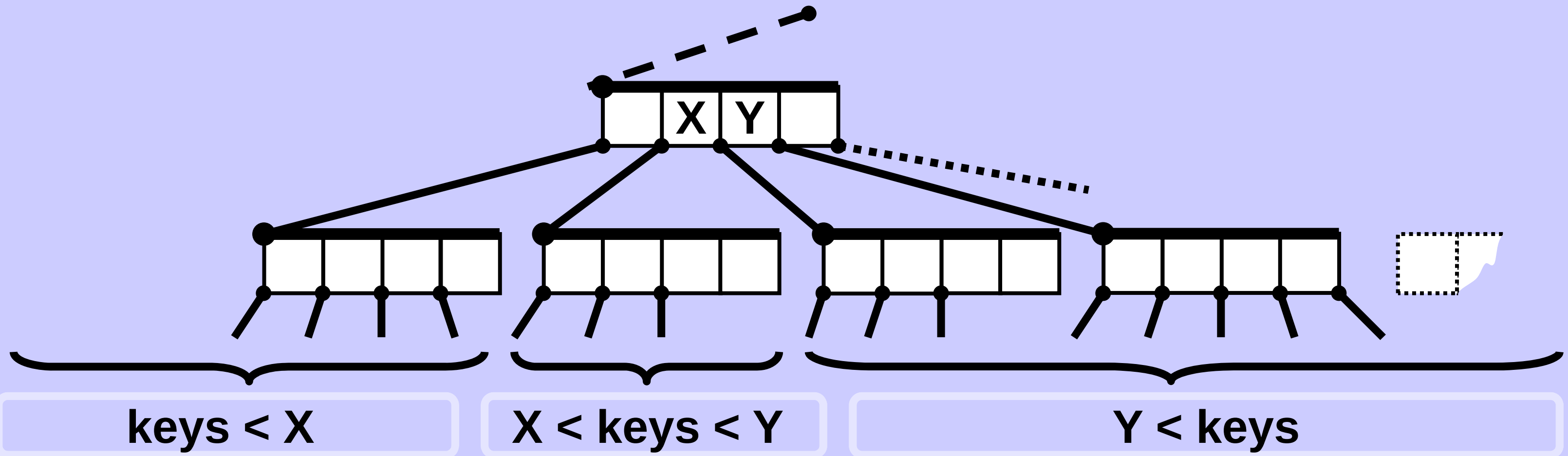
All lengths of paths from the root to the leaves are equal.
B-tree is perfectly balanced.

Keys in the nodes are kept sorted.

Fixed k > 1 dictates the same size of all nodes.

Each node except for the root contains at least k and at most 2k keys and if it is not a leaf it has at least k+1 and at most 2k+1 children.

The root may contain any number of keys from 1 to 2k.
If it is not simultaneously a leaf it has at least 2 and at most 2k+1 children.



B-tree -- Rudolf Bayer, Edward M. McCreight, 1972

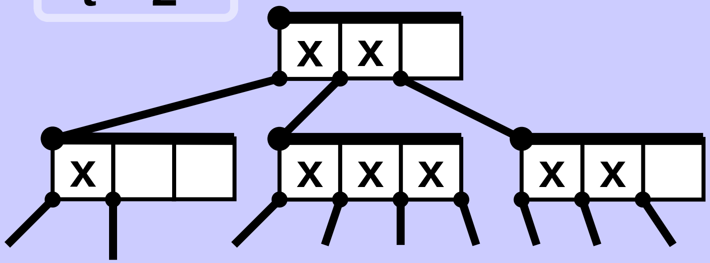
Cormen et al. 1990: B-tree degree:

Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree:

**a. Every node other than the root must have at least $t-1$ keys.
Every internal node other than the root thus has at least t children.
If the tree is nonempty, the root must have at least one key.**

**b. Every node may contain at most $2t-1$ keys.
Therefore, an internal node may have at most 2t children.**

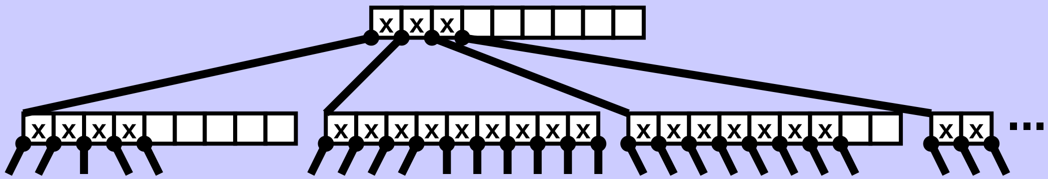
t = 2



min keys = 1
children = 2

max keys = 3
children = 4

t = 5



min keys = 4
children = 5

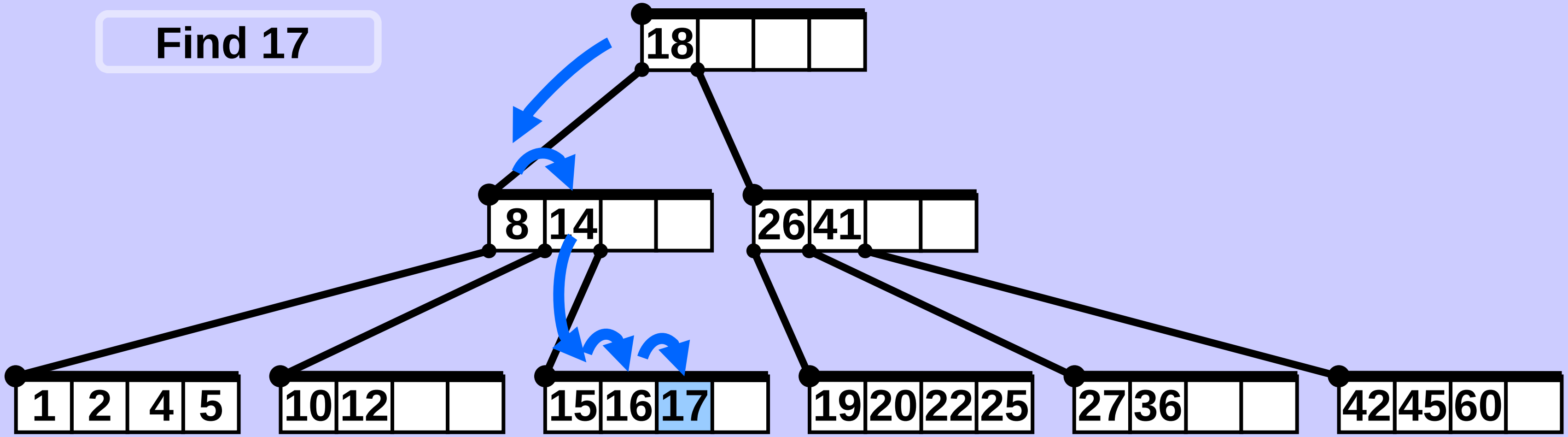
max keys = 9
children = 10

B-tree -- Update strategies

1. Multi phase strategy: "Solve the problem when it appears".
   First insert or delete the item and only then rearrange the tree if necessary.
   This may require additional traversing up to the root.

2. Single phase strategy: "Avoid the future problems".
   Travel from the root to the node/key which is to be inserted or deleted and during the travel rearrange the tree to prevent the additional traversing up to the root.

B-tree -- Find



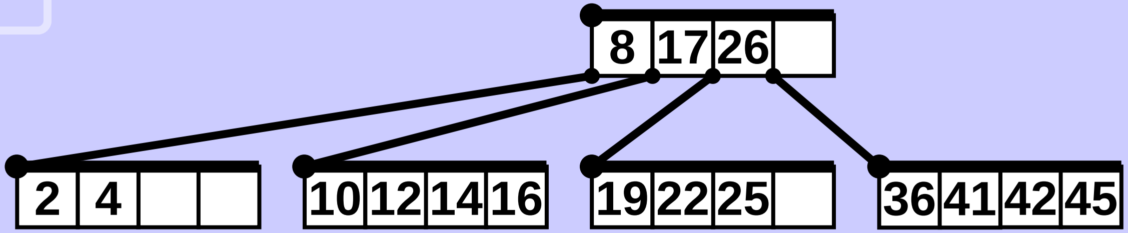
Search in the node is sequential (or binary or other...).

If the node is not a leaf and the key is not in the node then the search continues in the appropriate child node.

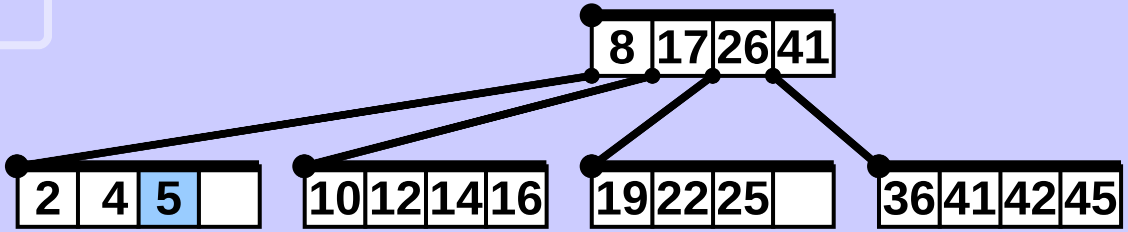
If the node is a leaf and the key is not in the node then the key is not in the tree.

B-tree -- Insert

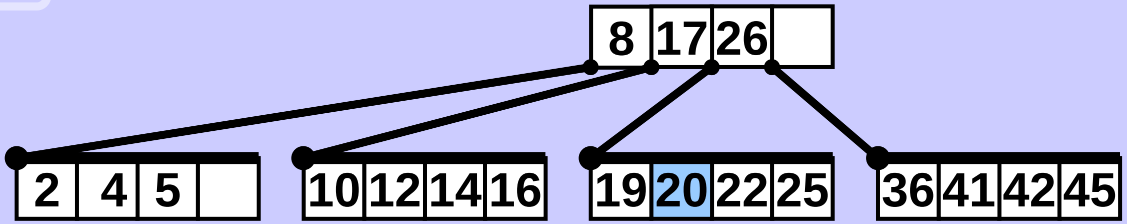
B-tree



Insert 5

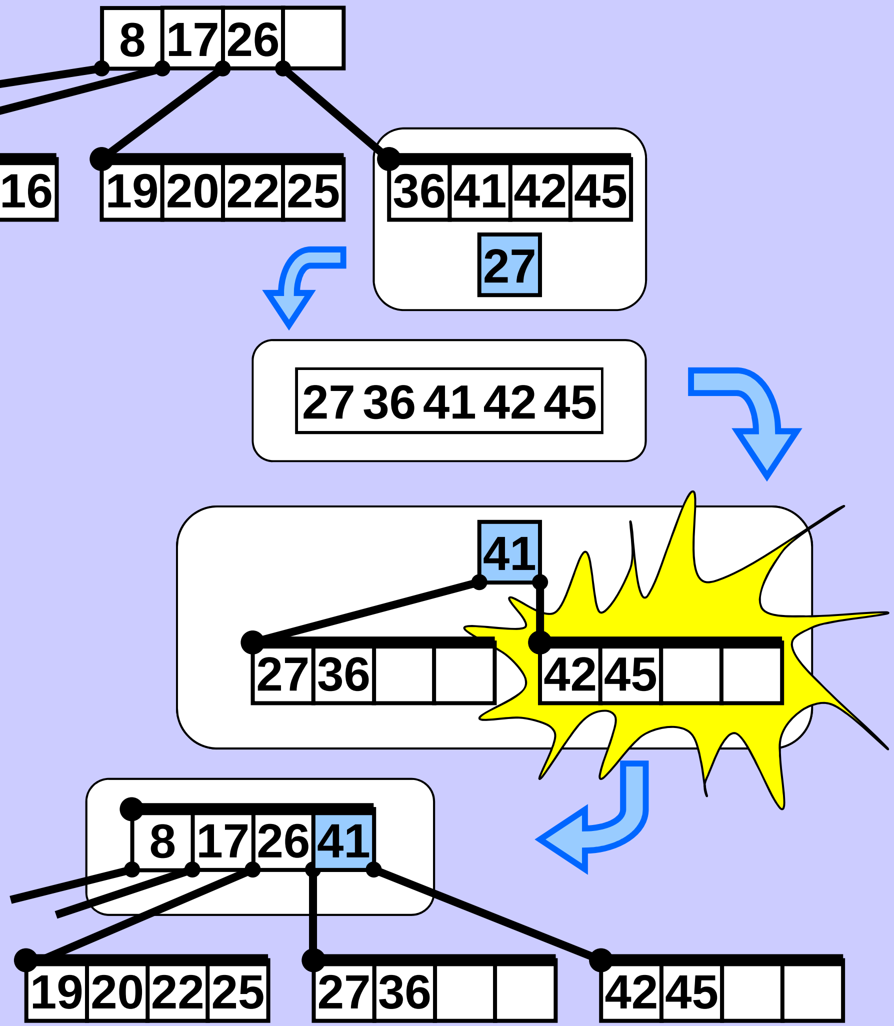


Insert 20



B-tree -- Insert

Insert 27



Sort outside the tree.

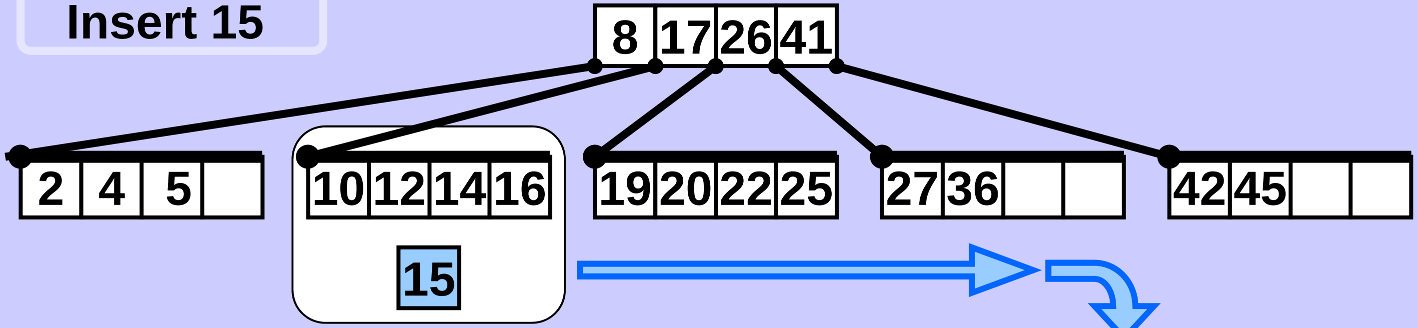
Select median, create new node, move to it the values bigger than the median.

Try to insert the median into the parent node.

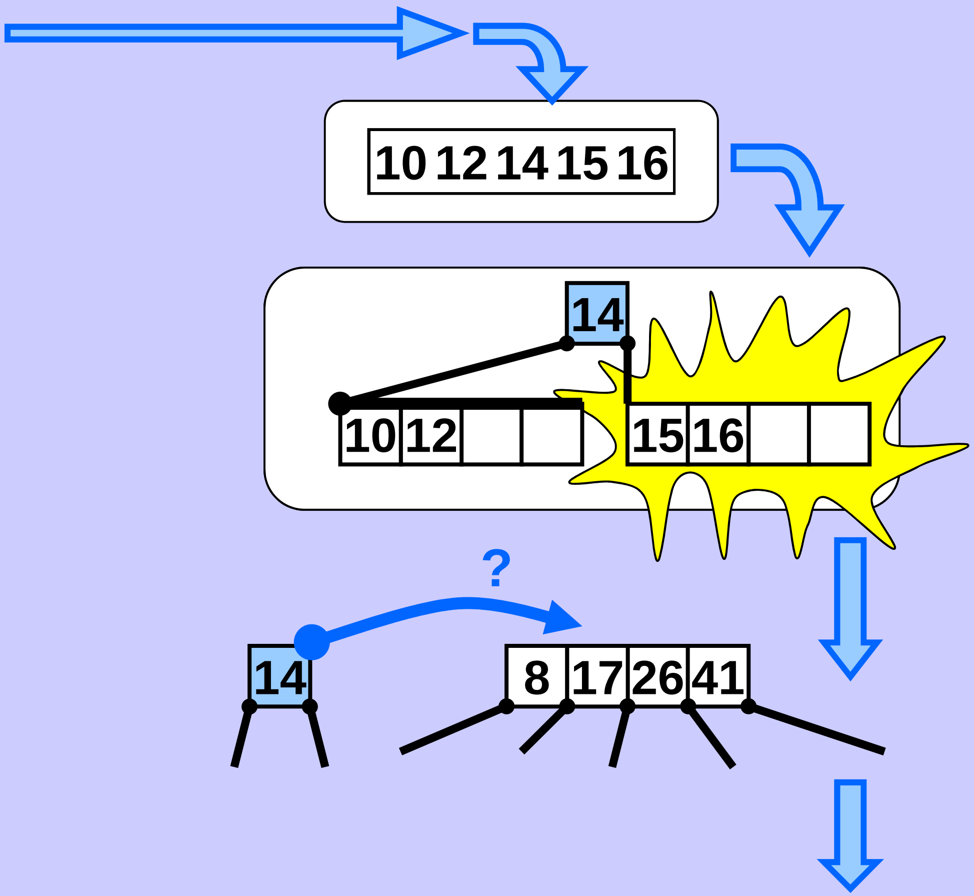
Success.

B-tree -- Insert

Insert 15



Sort outside the tree.



Select median,
create new node,
move to it the values
bigger than the median.

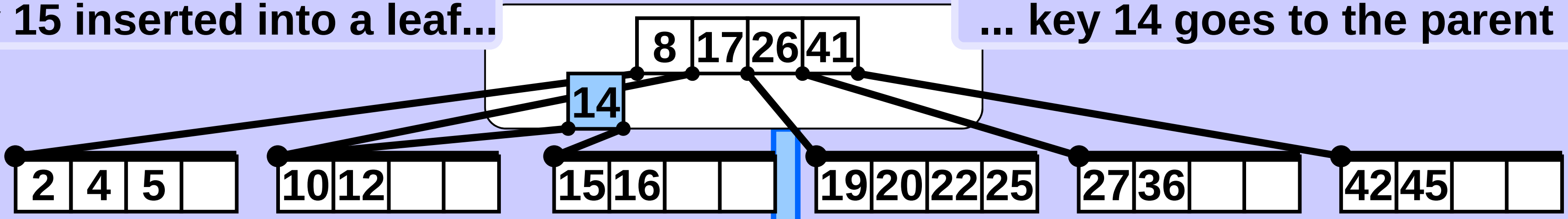
Try to insert the median
into the parent node.

Success? ...

B-tree -- Insert

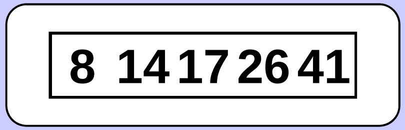
Key 15 inserted into a leaf...

... key 14 goes to the parent

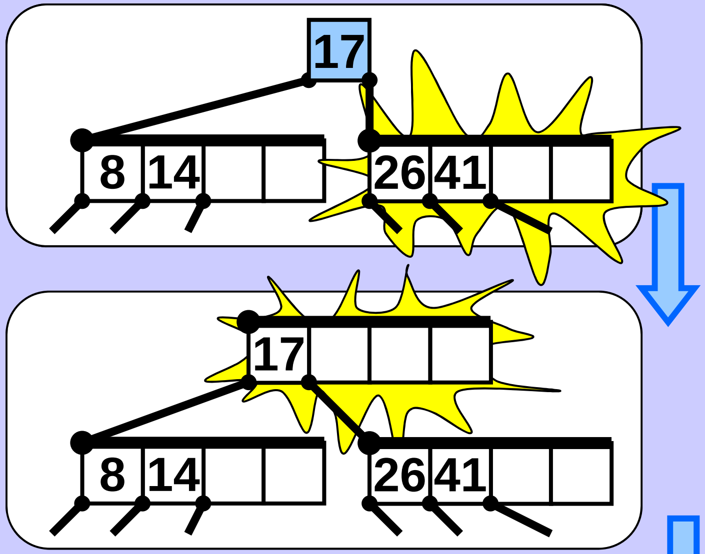


The parent node is full – iterate the process.

Sort values



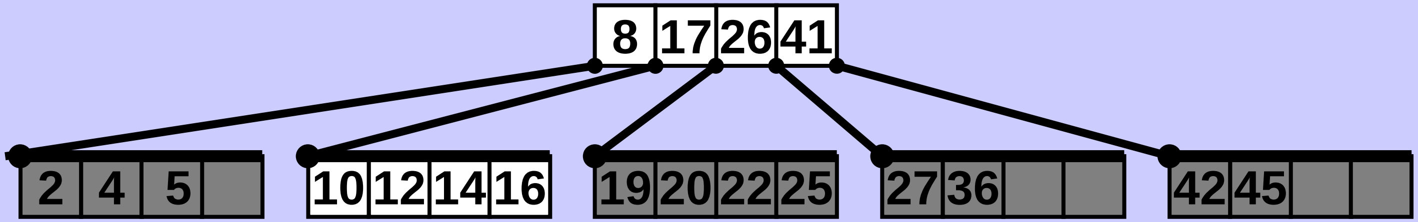
Select median, create new node, move to it the values bigger than the median together with the corresponding references.



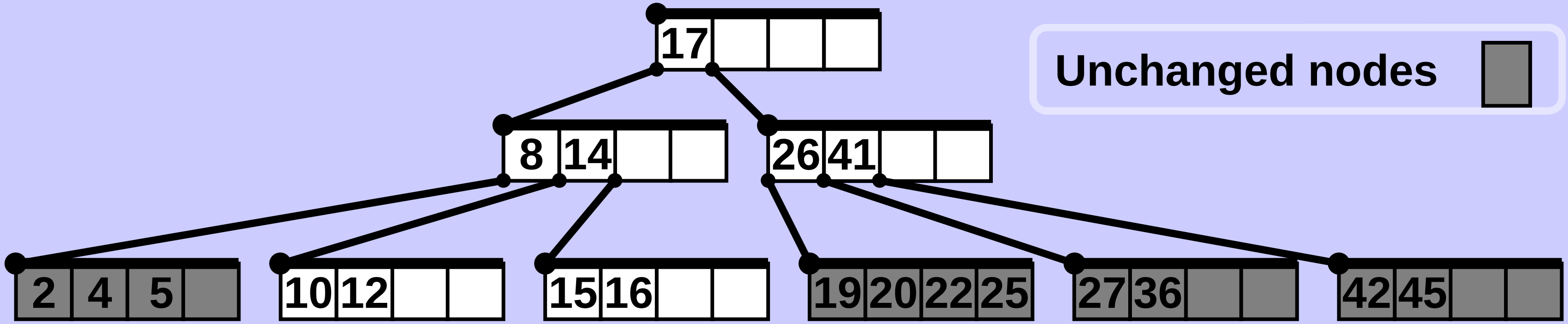
Cannot propagate the median into the parent (there is no parent), create a new root and store the median there.

B-tree -- Insert

Recapitulation - insert 15



Insert 15

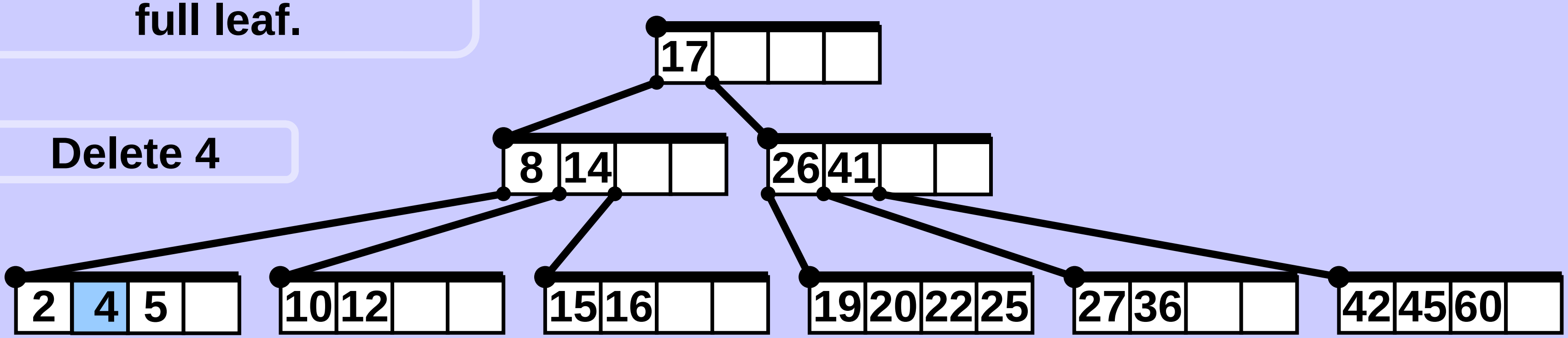


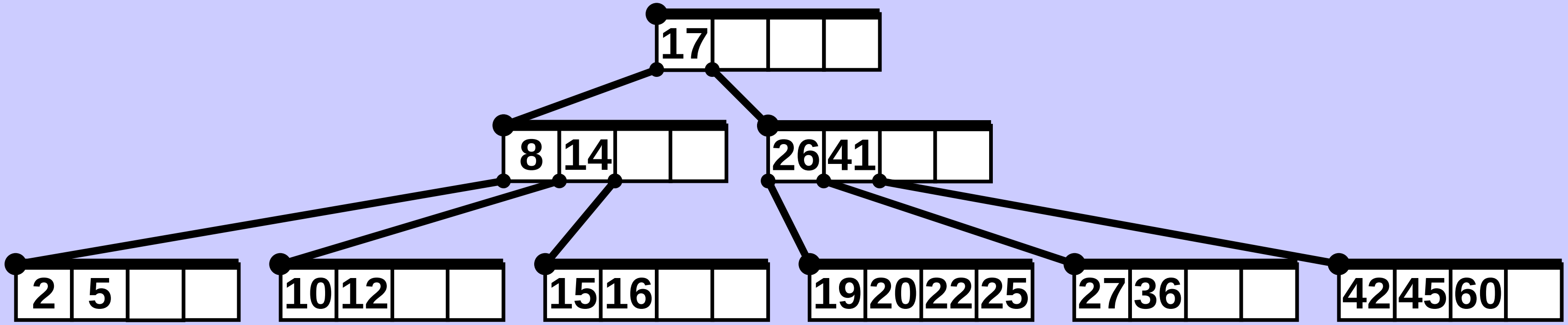
Each level acquired one new node, a new root was created, the tree grows upwards and remains perfectly balanced.

B-tree -- Delete

Delete in a sufficiently full leaf.

Delete 4



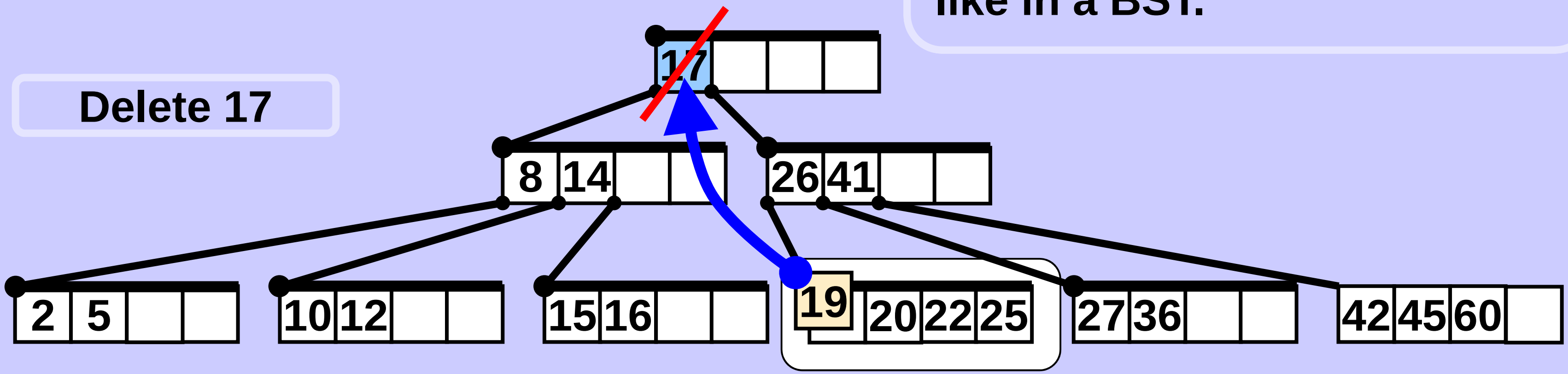


B-tree -- Delete

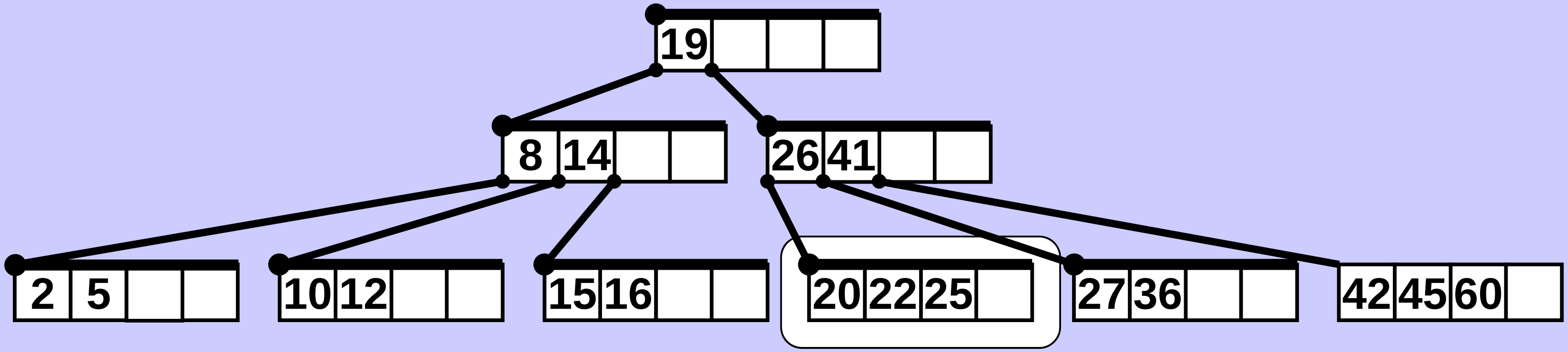
Delete in an internal node

The deleted key is substituted by the smallest bigger key, like in a BST.

Delete 17



The smallest bigger key is always in the leaf in a B-tree.
If the leaf is sufficiently full the delete operation is complete.

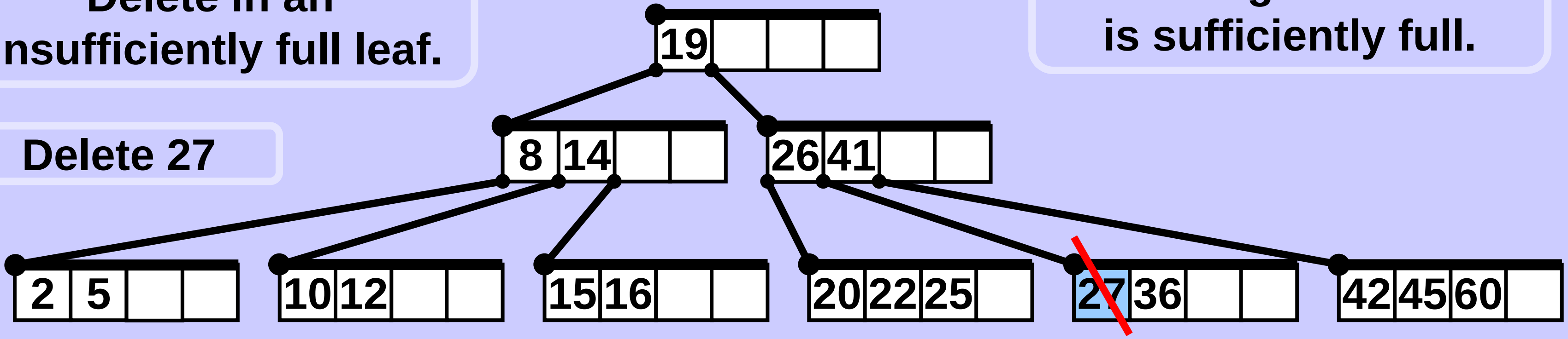


B-tree -- Delete

Delete in an insufficiently full leaf.

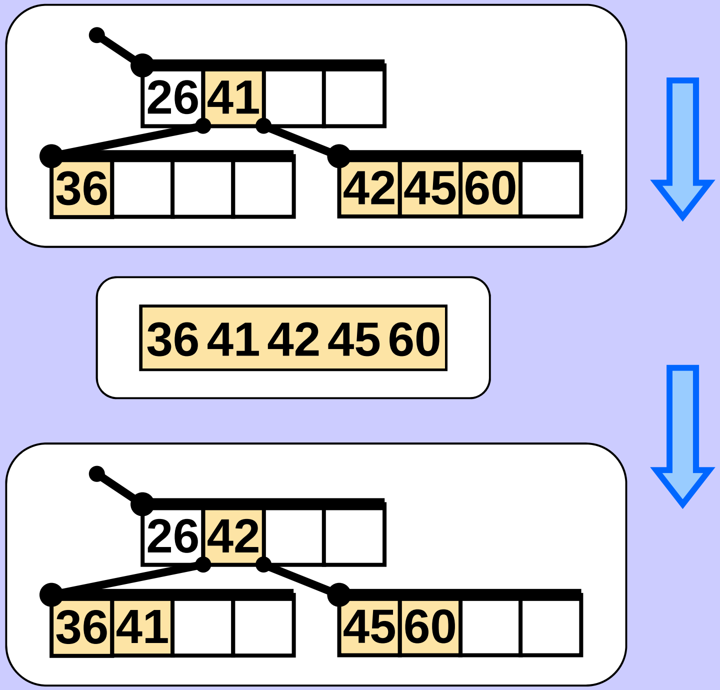
The neighbour leaf is sufficiently full.

Delete 27



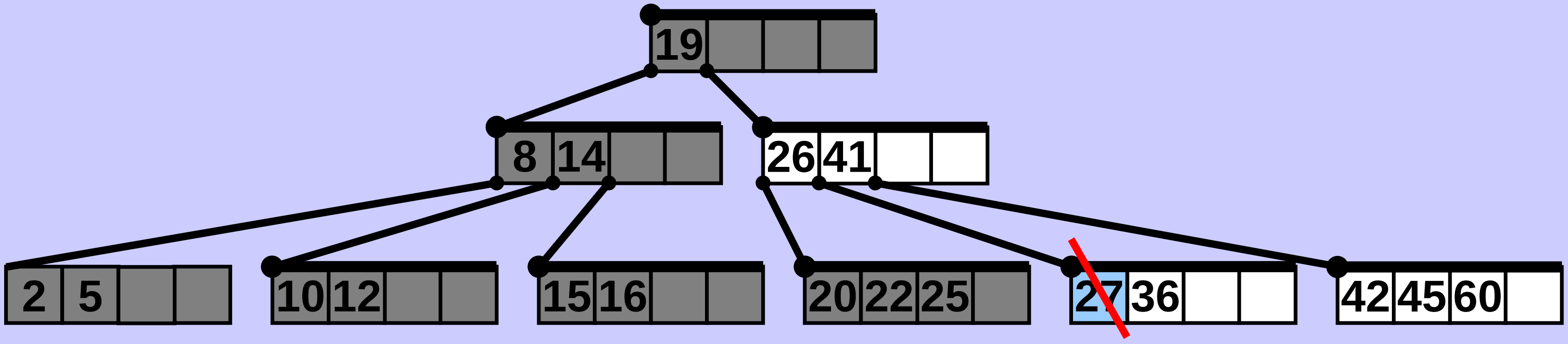
Merge the keys of the two leaves with the dividing key in the parent into one sorted list.

Insert the median of the sorted list into the parent and distribute the remainig keys into the left and right children of the median.



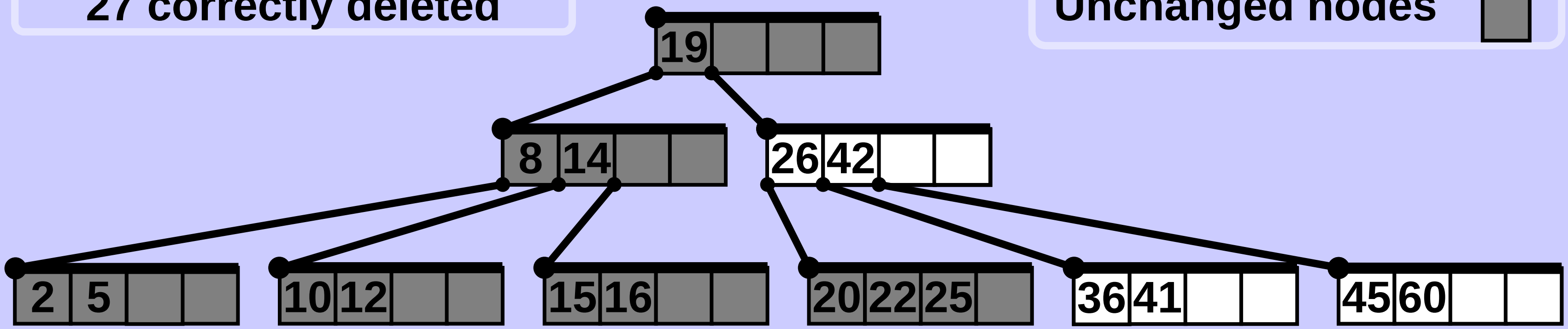
B-tree -- Delete

Recapitulation - delete 27



27 correctly deleted

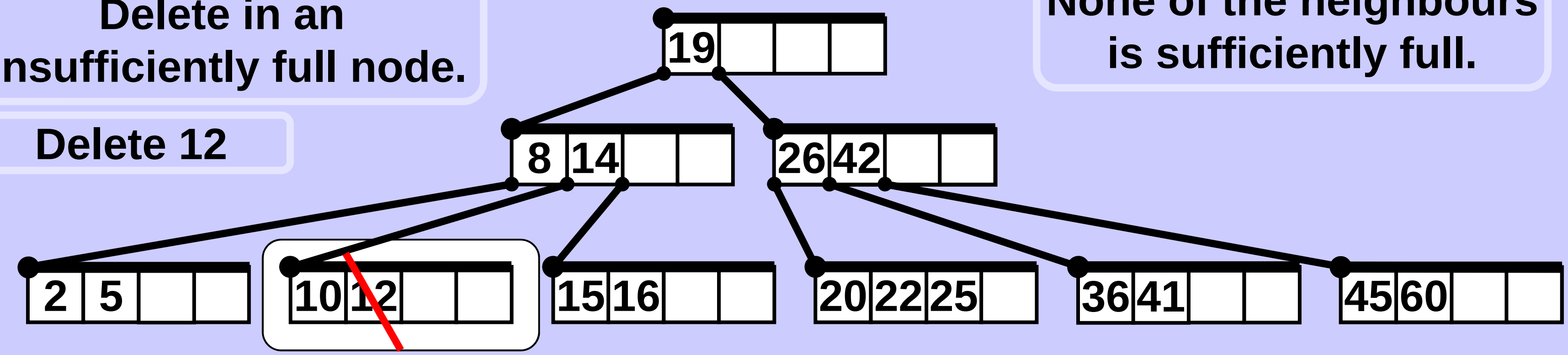
Unchanged nodes



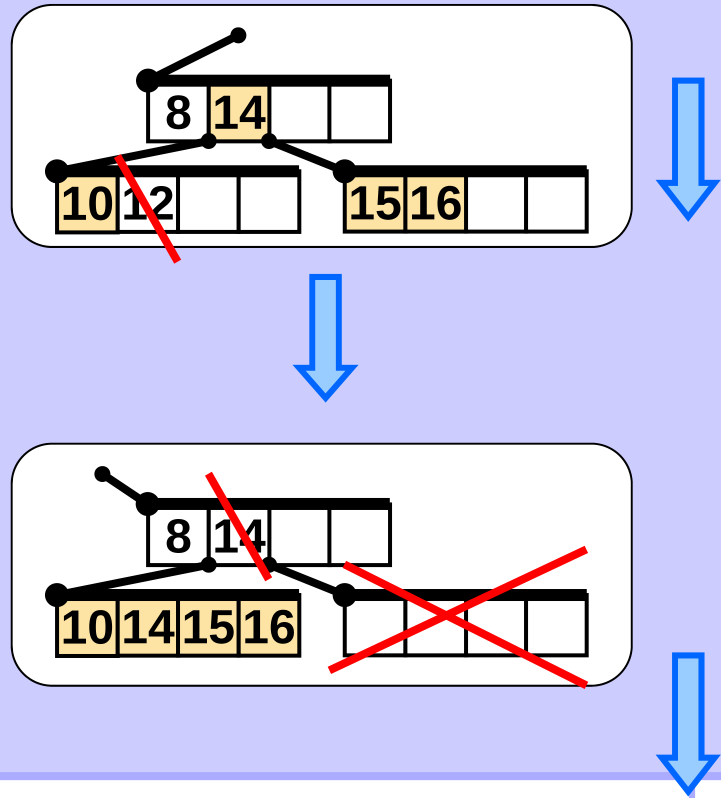
B-tree -- Delete

Delete in an insufficiently full node.

Delete 12



None of the neighbours is sufficiently full.

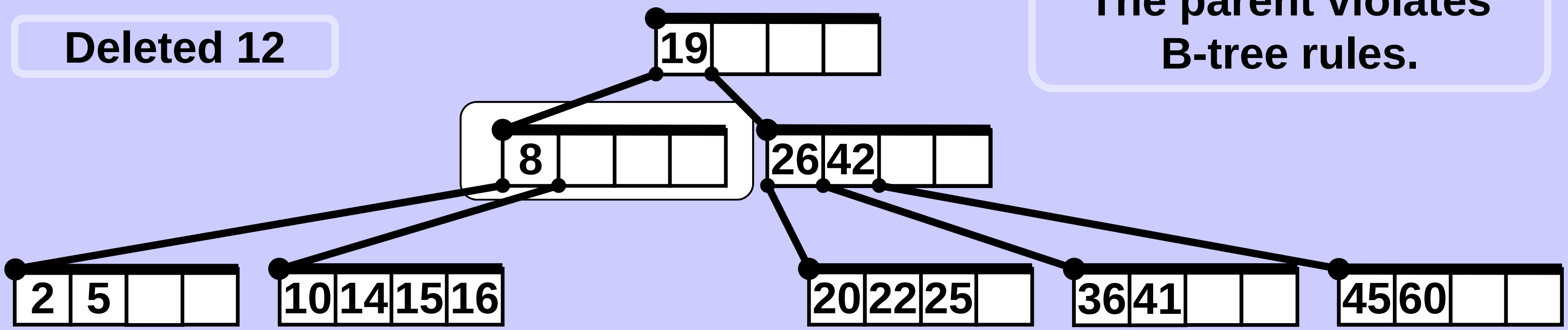


Merge the keys
  of the node and
  of one of the neighbours and
  the median in the parent
into one sorted list.
Move all these keys to the original node,
delete the neighbour,
remove the original median and
associated reference from the parent.

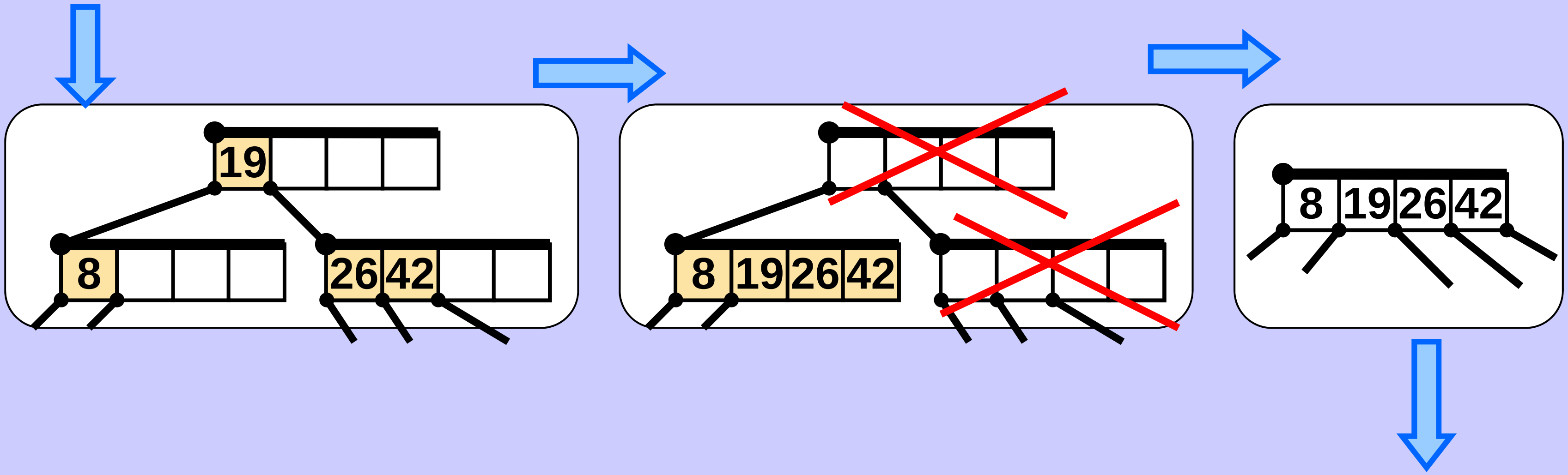
B-tree -- Delete

Deleted 12

The parent violates B-tree rules.

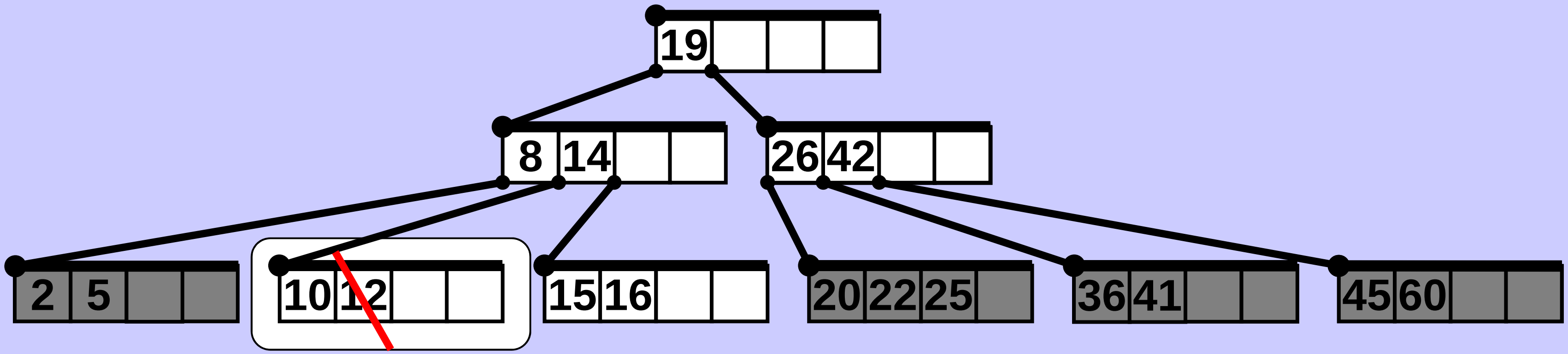


If the parent of the deleted node is not sufficiently full apply the same deleting strategy to the parent and continue the process towards the root until the rules of B-tree are satisfied.



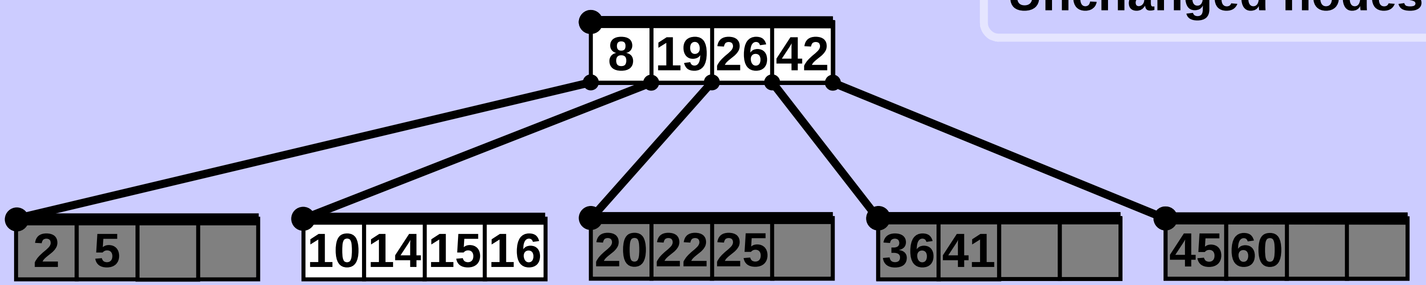
B-tree -- Delete

Recapitulation - delete 12



Key 12 was deleted and the tree was reconstructed accordingly.

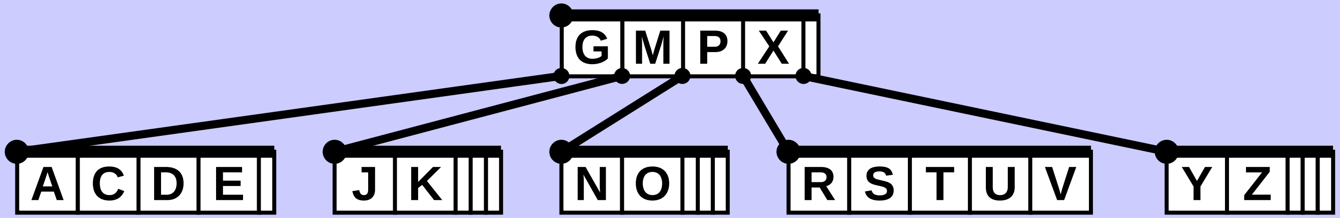
Unchanged nodes



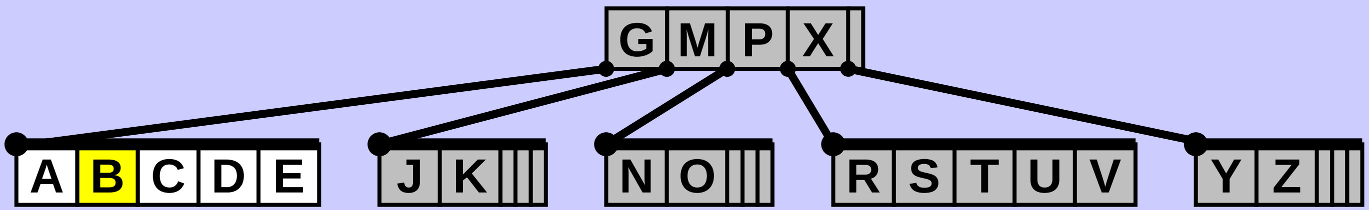
B-tree -- Insert

Single phase strategy

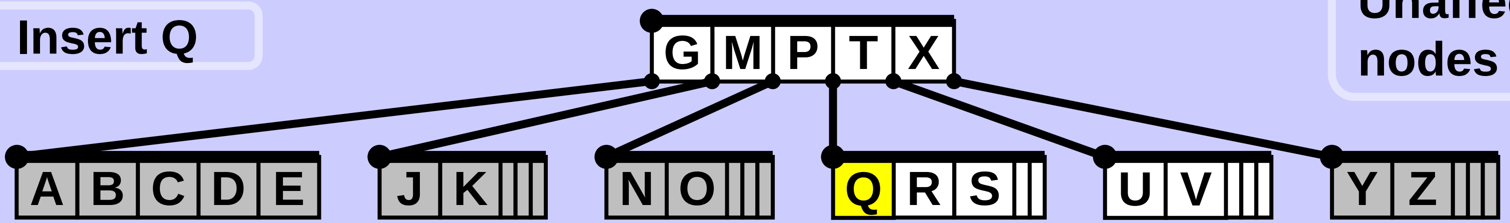
[CLRS, *Cormen et al. 1990*], t = 3, minimum degree 3, max degree = 6, minimum keys in node = 2, maximum keys in node = 5.



Insert B



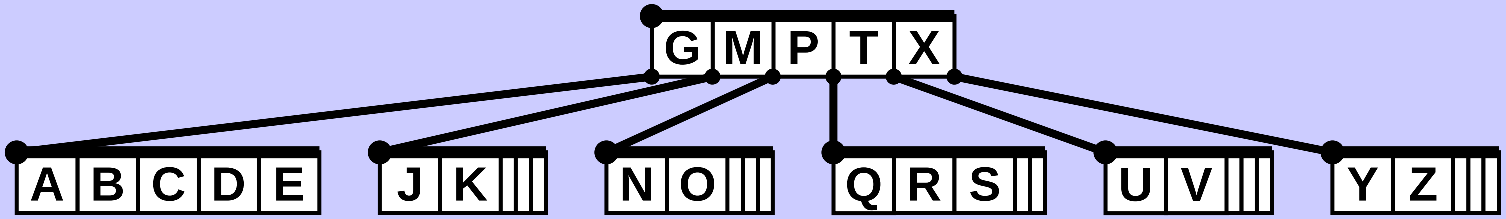
Insert Q



Unaffected nodes

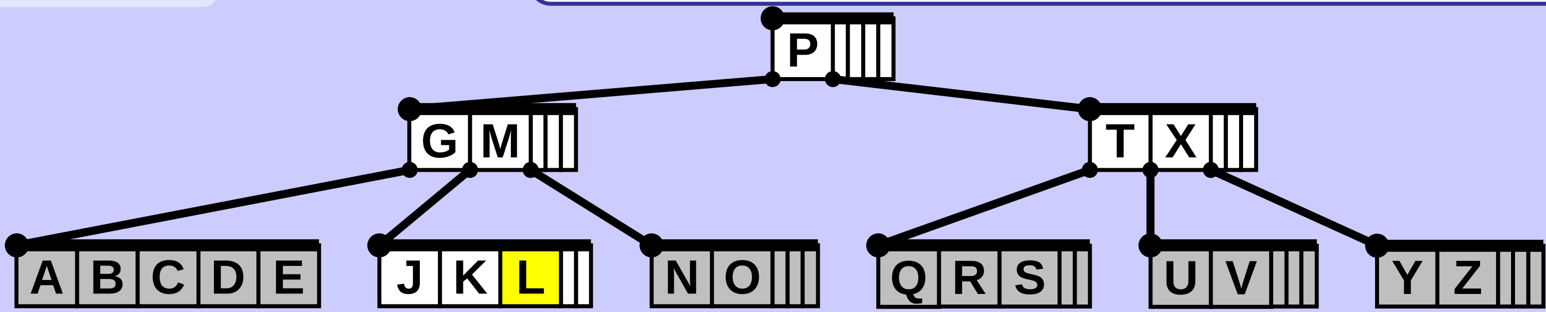
B-tree -- Insert

Single phase strategy



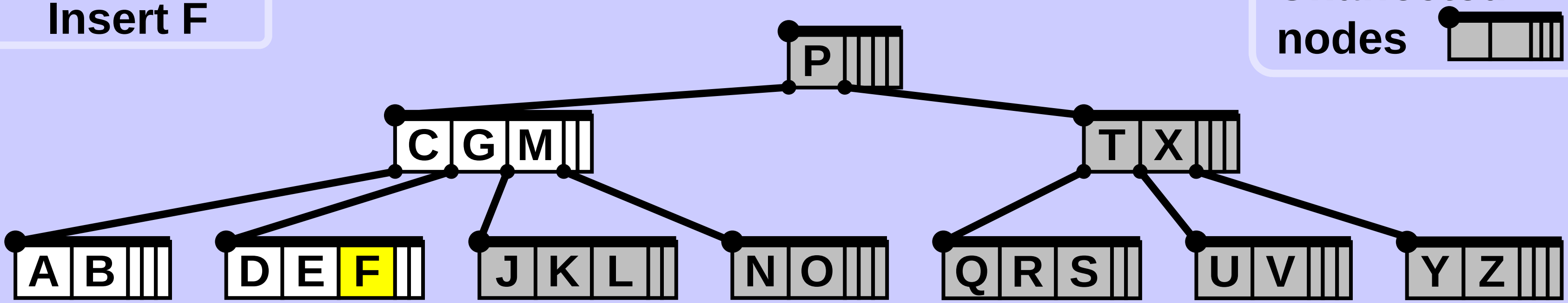
Insert L

Single phase: Split the root, because it is full, and then continue downwards inserting L



Insert F

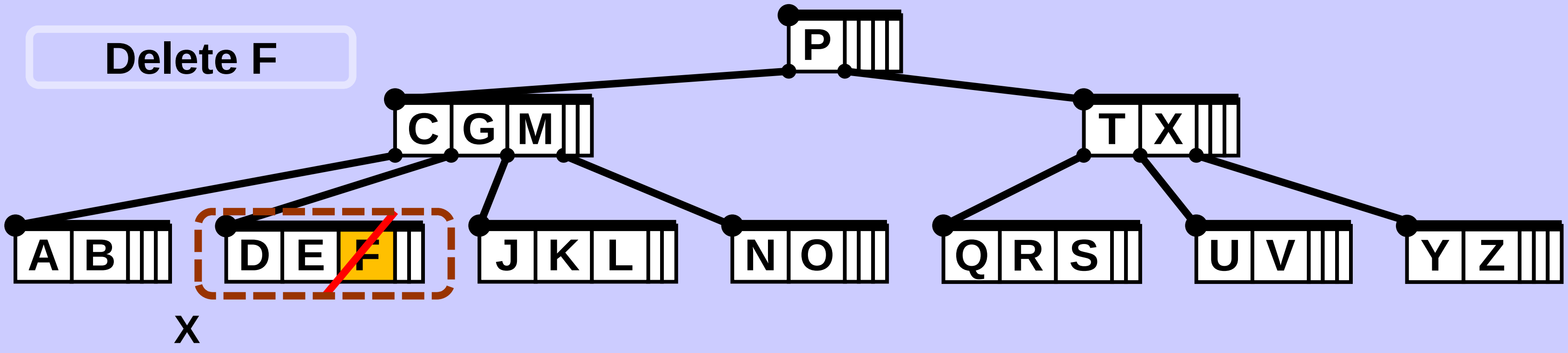
Unaffected nodes



B-tree -- Delete

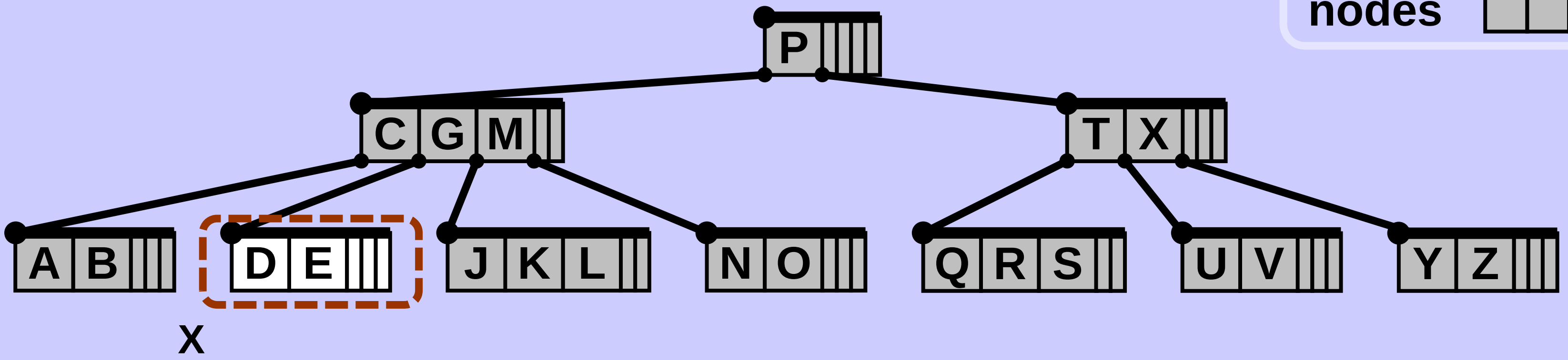
Single phase strategy

Delete F



1. Key **k** is in node **X** and **X** is a leaf, delete the key **k** from **X**. Stop.

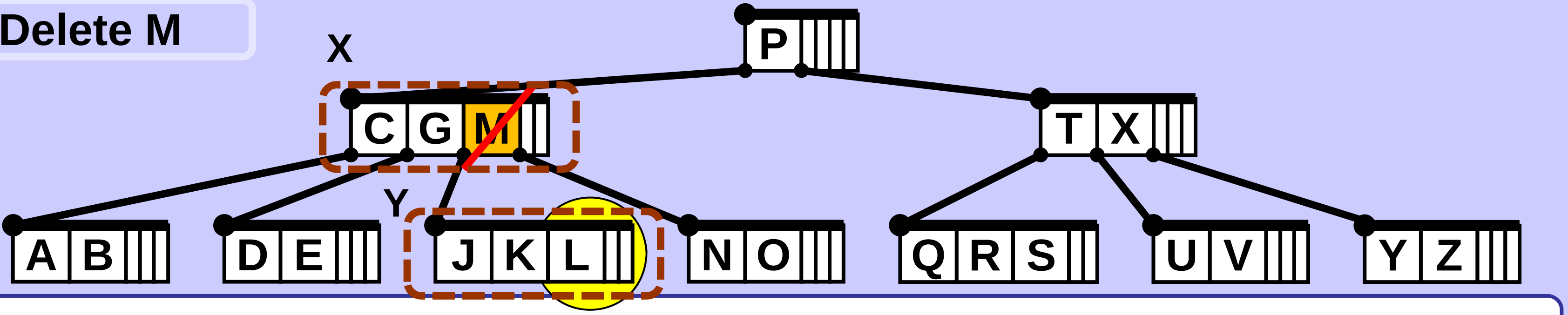
Unaffected nodes



B-tree -- Delete

Single phase strategy

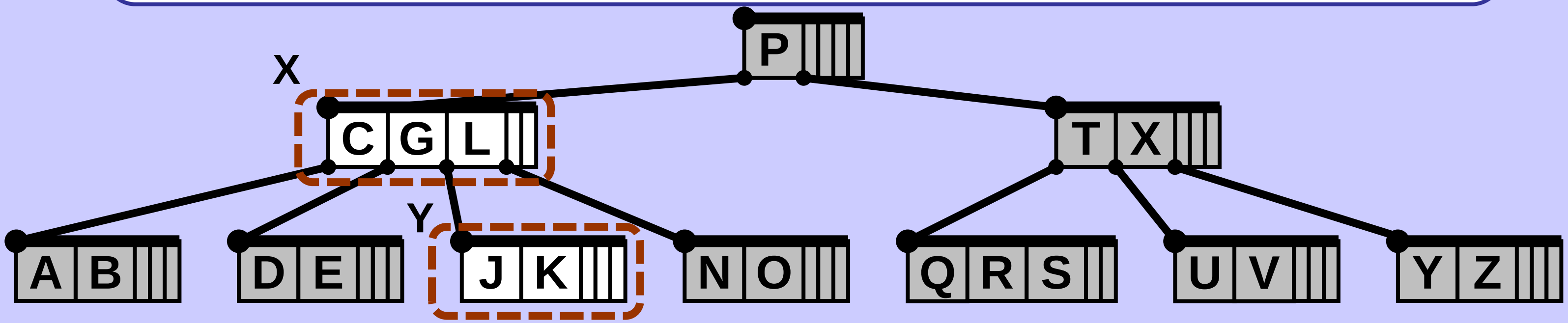
Delete M



2. Key **k** is in node **X** and **X** is an internal node. -- 2a. or 2b. or 2c.

2a. Child **Y** that precedes **k** in node **X** has at least t keys.
Find the predecessor $\mathbf{k_p}$ of **k** in the subtree rooted at **Y**.
Recursively delete $\mathbf{k_p}$, and replace **k** by $\mathbf{k_p}$ in **X**.
(We can find $\mathbf{k_p}$ and delete it in a single downward pass.)

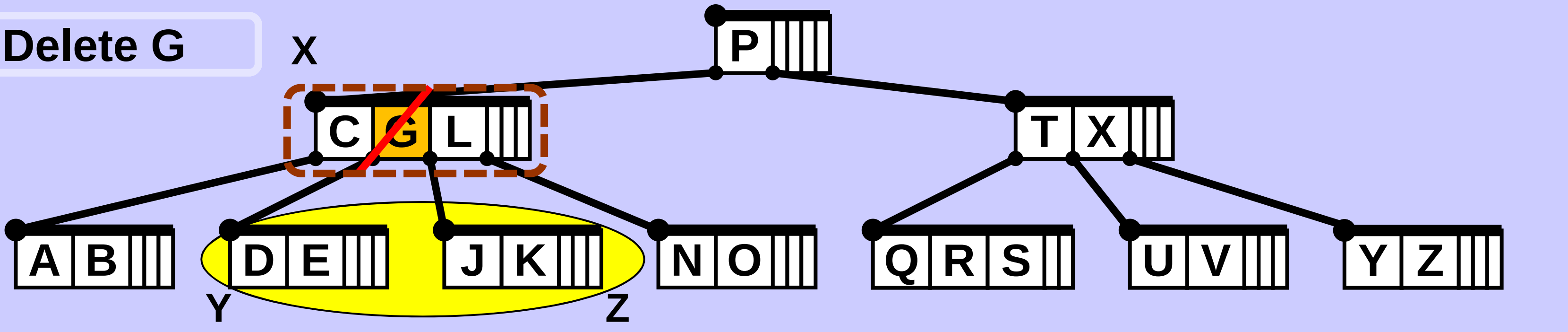
2b. Child **Z** that follows **k** in node **X** has at least t keys.
Mirror-symmetric to 2a.



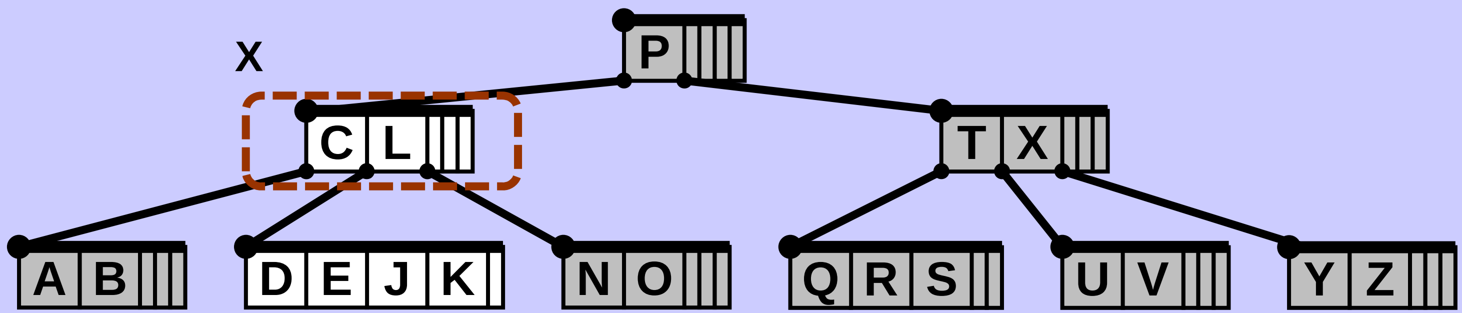
B-tree -- Delete

Single phase strategy

Delete G



2c. Both **Y** and **Z** have only t–1 keys.
Merge **k** and all of **Z** into **Y**, so that **X** loses both **k** and the pointer to **Z**, and **Y** now contains 2t–1 keys.
Free **Z** in the memory and recursively delete **k** from **Y**.



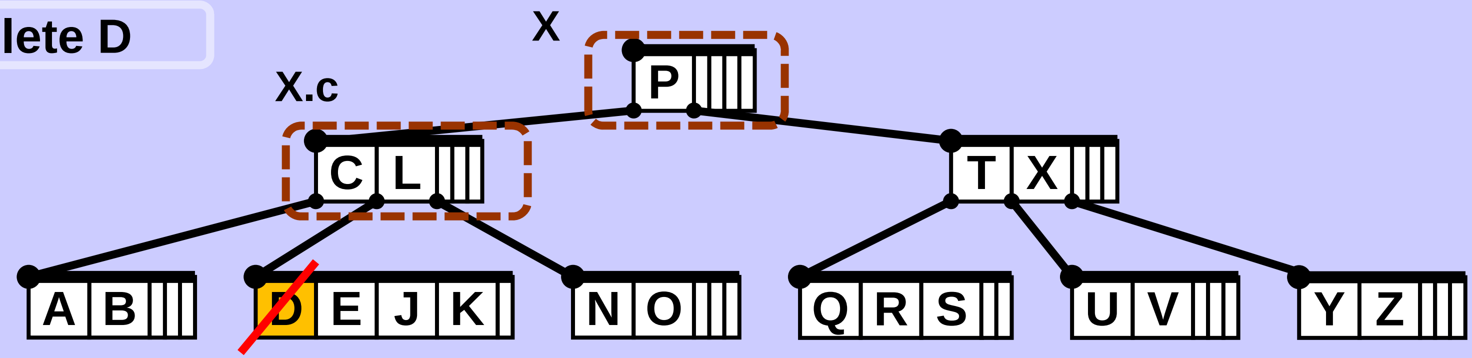
B-tree -- Delete

Single phase strategy

3. Key **k** is not present in internal node **X**.
   Determine the child **X.c** of **X**.
   X.c is a root of such subtree that contains **k**, if **k** is in the tree at all.
   If **X.c** has only t–1 keys, execute step 3a. or 3b. as necessary,
   to guarantee that we descend to a node containing at least t keys.

Then continue by recursing (and deleting **k**) on the appropriate child of **X**.

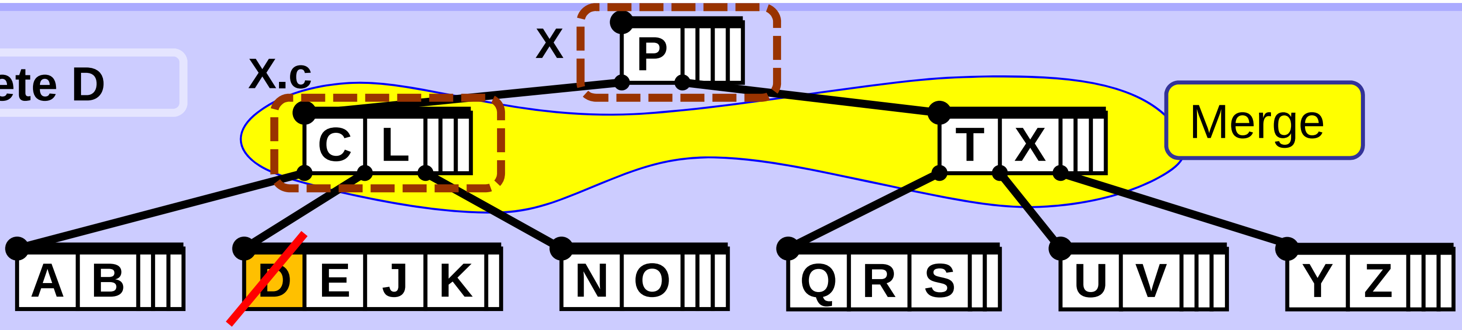
Delete D



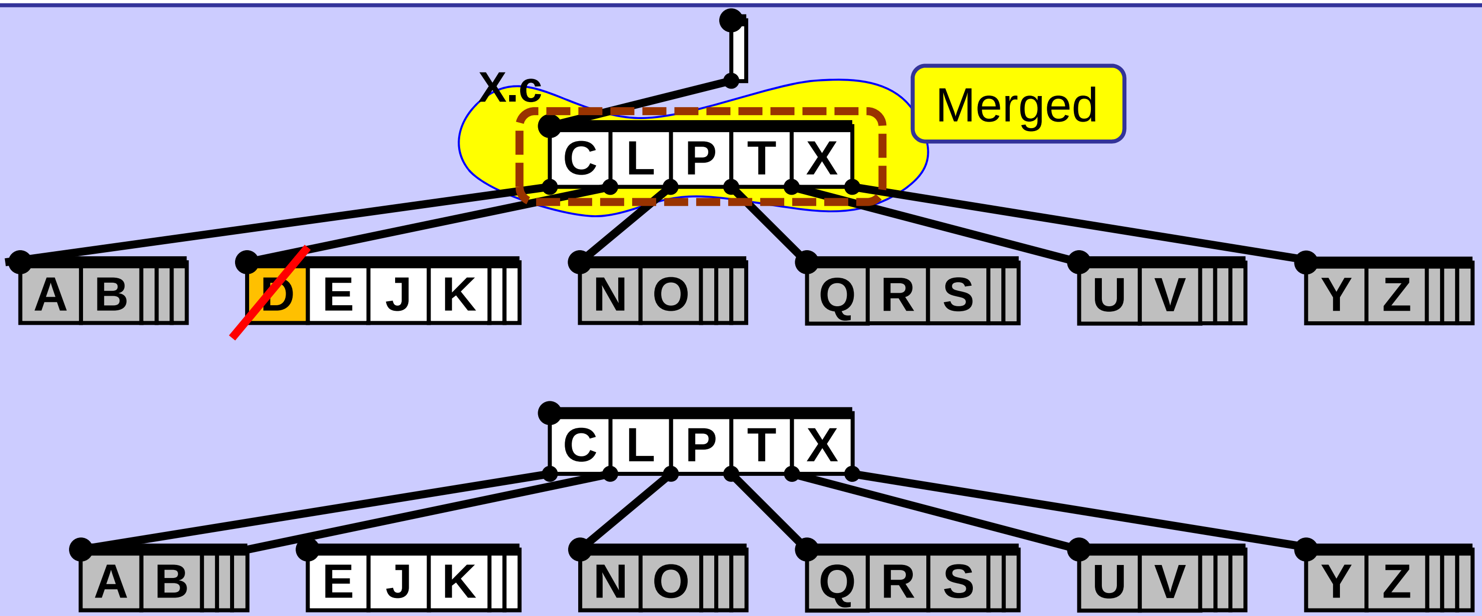
B-tree -- Delete

Single phase strategy

Delete D



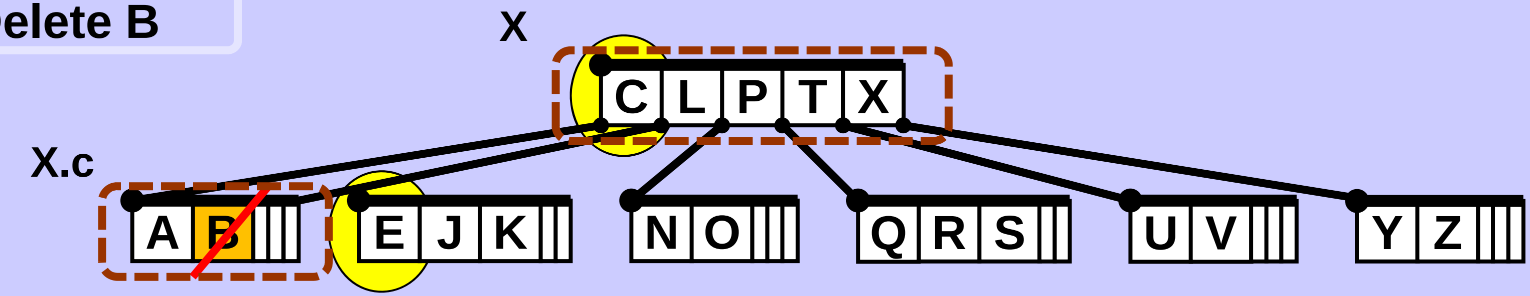
3a. If **X.c** and both of **X.c** 's immediate siblings have t–1 keys,
merge **X.c** with one sibling,
which involves moving a key from **X** down into the new merged node
to become the median key for that node.



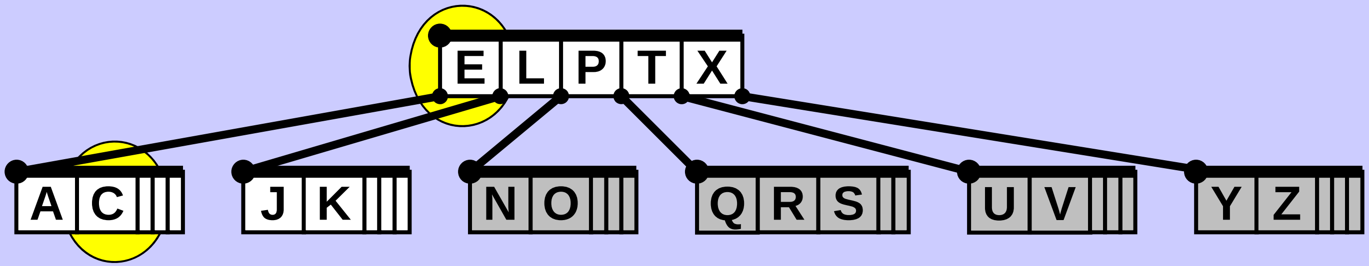
B-tree -- Delete

Single phase strategy

Delete B



3b. If **X.c** has only t–1 keys but has an immediate sibling with at least t keys, give **X.*c*** an extra key by moving a key from **X** down into **X.c**, moving a key from**X.c** ’s immediate left or right sibling up into **X**, and moving the appropriate child pointer from the sibling into **X.c**.



B-tree -- asymptotic complexities

Find	$O(b \cdot \log_b n)$
Insert	$\Theta(b \cdot \log_b n)$
Delete	$\Theta(b \cdot \log_b n)$

n is the number of keys in the tree, b is the branching factor, i.e. the order of the tree, i.e. the maximum number of children of a node.

Note: Be careful, some authors (e.g. CLRS) define degree/order of B-tree as [b/2], there is no unified precise common terminology.