FIT2004 - Algorithms and Data Structures

Seminar 11 - Search Trees

Rafael Dowsley

19 May 2025

Agenda

Divide-and-Conquer (W1-3)	Greedy Algorithms (W4-5)	Dynamic Programming (W6-7)	Network Flow (W8-9)	Data Structures (W10-11)

1. Binary Search Trees
2. AVL Trees
3. 2-3 Search Trees
4. Red-Black Trees

SETU

- Please fill SETU (Student Evaluation of Teaching and Units) survey!
- `https://monash.bluera.com/monash`

Lookup table

- A lookup table allows inserting, searching and deleting values by their keys.
- The idea of a lookup table is very general and important in information processing systems.
- The database that Monash University maintains on students is an example of a table. This table might contain information about:
  - Student ID
  - Authcate
  - First name
  - Last name
  - Course(s) enrolled

Lookup table

- Elements of a table normally contain a key plus some other attributes or data.
- The key is something unique that unambiguously identifies an element. It might be a number or a string (e.g., Student ID or authcate).
- Elements can be looked up (or searched) using this key.

Sorting based lookup

- Keep the n elements sorted on their keys in an array.
- **Searching:** $\Theta(\log n)$ using binary search.
- **Insertion:** $\Theta(n)$.
  - Binary search to find the sorted location of new element – $\Theta(\log n)$.
  - Insert the new element and shift all larger elements toward right – $\Theta(n)$.
- **Deletion:** $\Theta(n)$.
  - Binary search to find the key – $\Theta(\log n)$.
  - Delete the key – $\Theta(1)$.
  - Shift all the larger elements to left – $\Theta(n)$.
- Is it possible to do better?

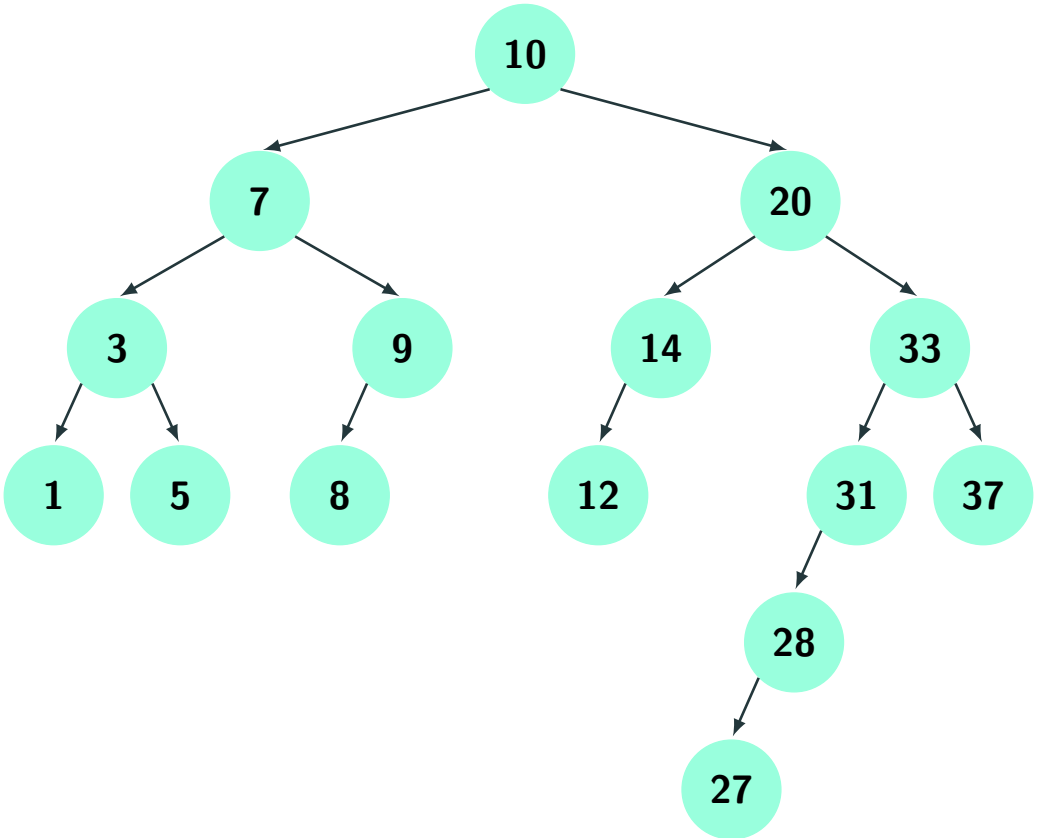
Binary Search Trees

Binary Search Trees

Binary Search Tree (BST)

- The empty tree is a BST.
- If the tree is not empty, it is an BST if:
  1. The root node contains exactly one key.
  2. The left subtree of the root is a BST and all its keys are smaller than the root key.
  3. The right subtree of the root is a BST and all its keys are greater than the root key.

BST example

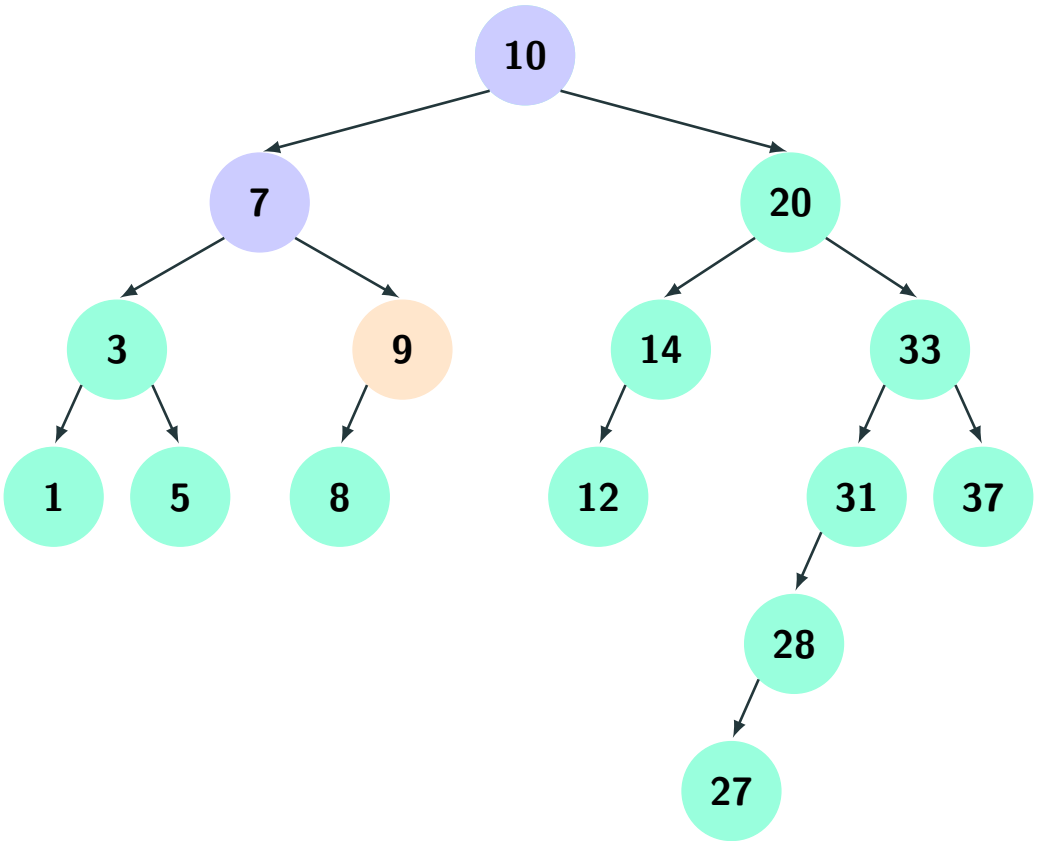


Searching in a BST

- If the BST is empty, the key is not found.
- Otherwise, compare the key k that is being searched with the root key r:
  1. If it is equal, the key is found.
  2. If $k < r$, proceed recursively with the left subtree.
  3. If $k > r$, proceed recursively with the right subtree.

Search example

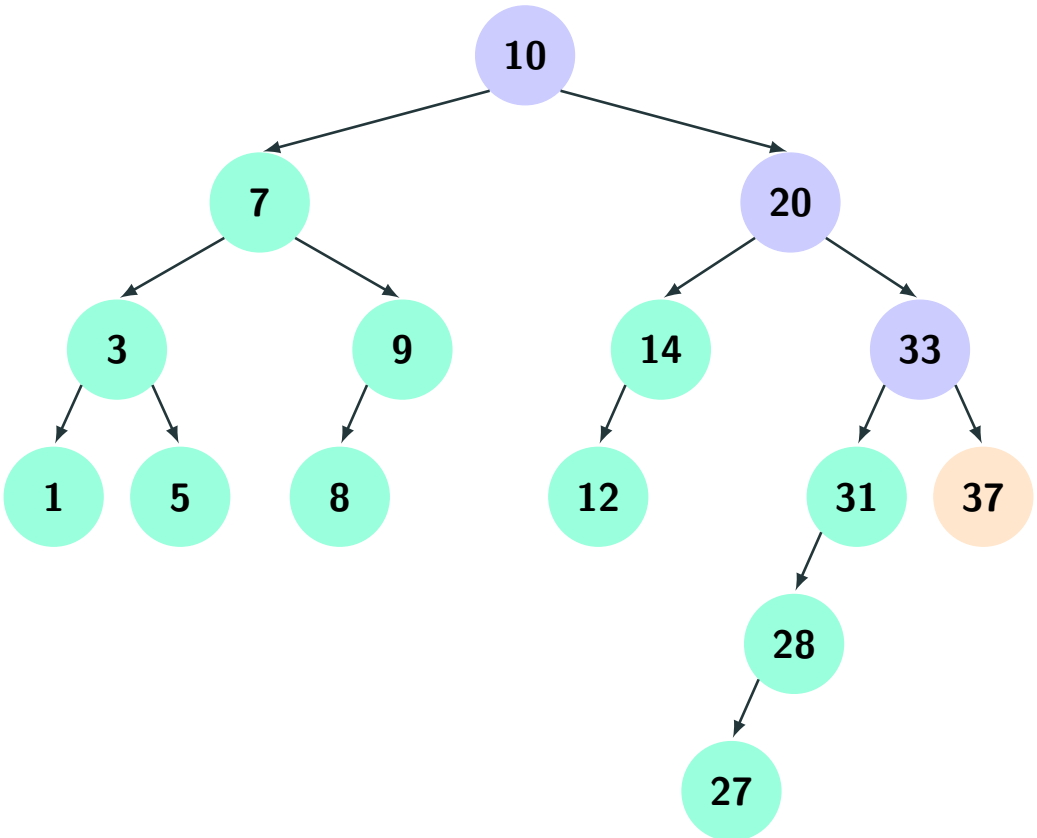
Searching for 9:



Key 9 was found.

Search example

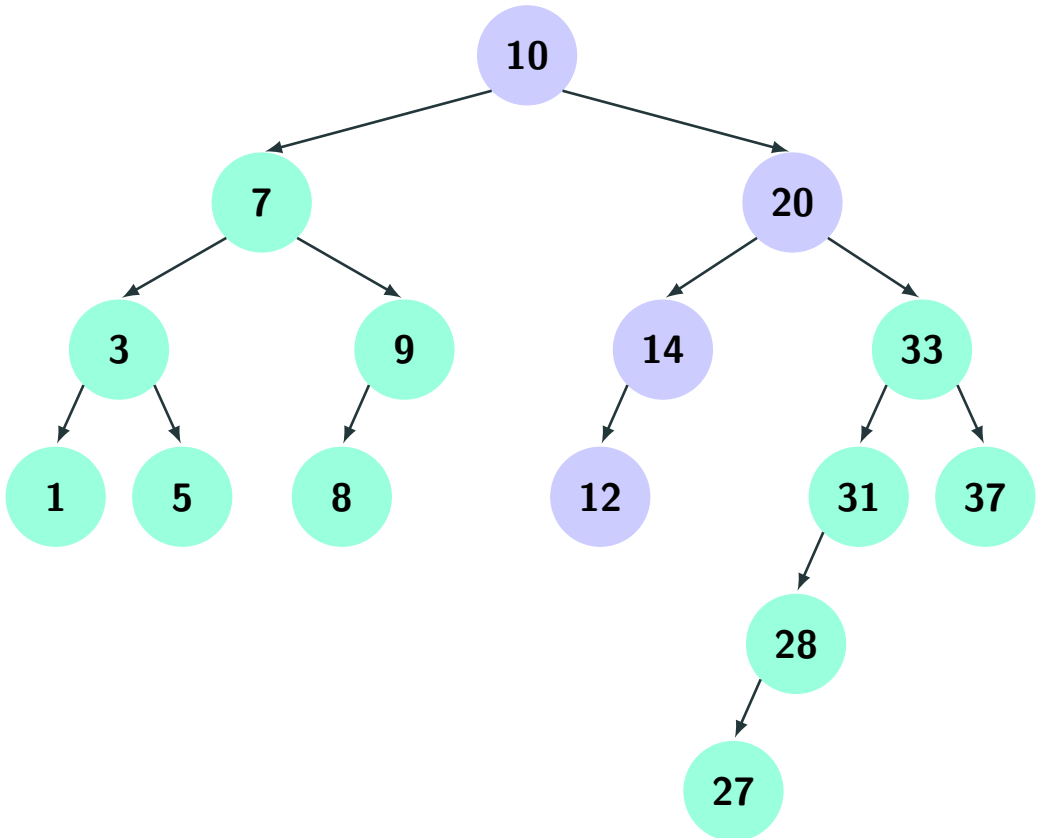
Searching for 37:



Key 37 was found.

Search example

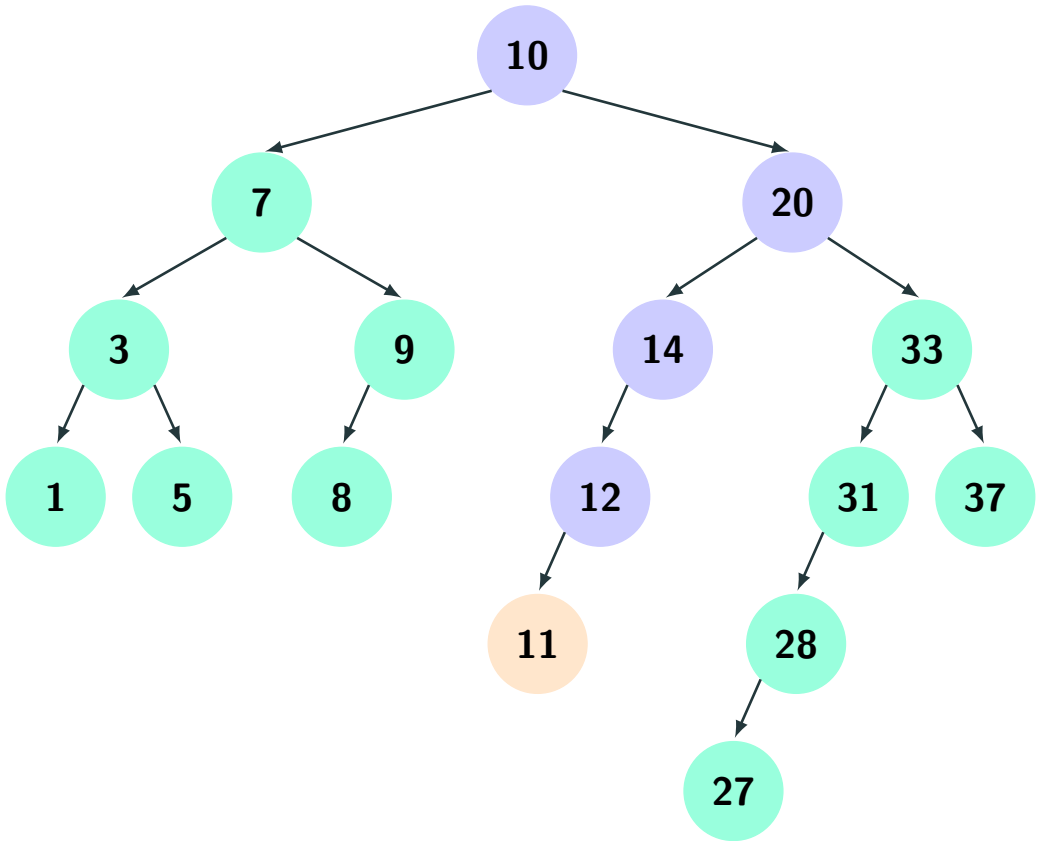
Searching for 11:



Key 11 not found.

Inserting a key

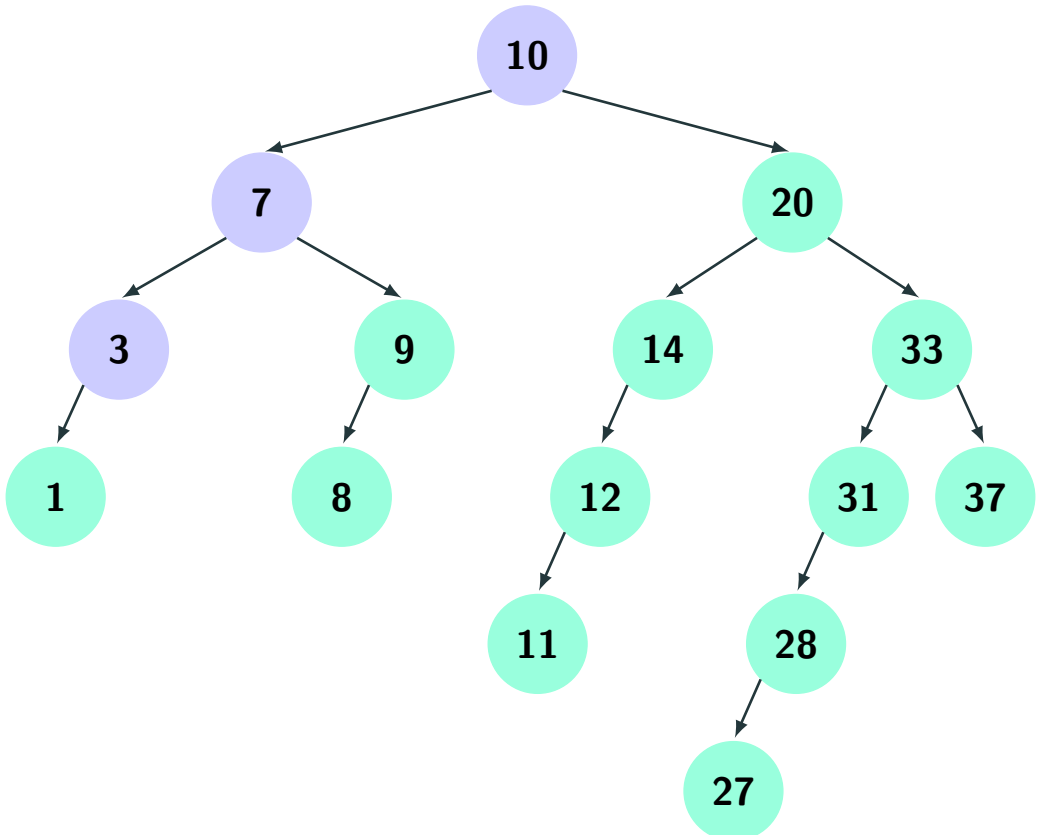
Search for the key. If not found, insert in the final location. E.g., key 11:



Key 11 inserted.

Deleting a key

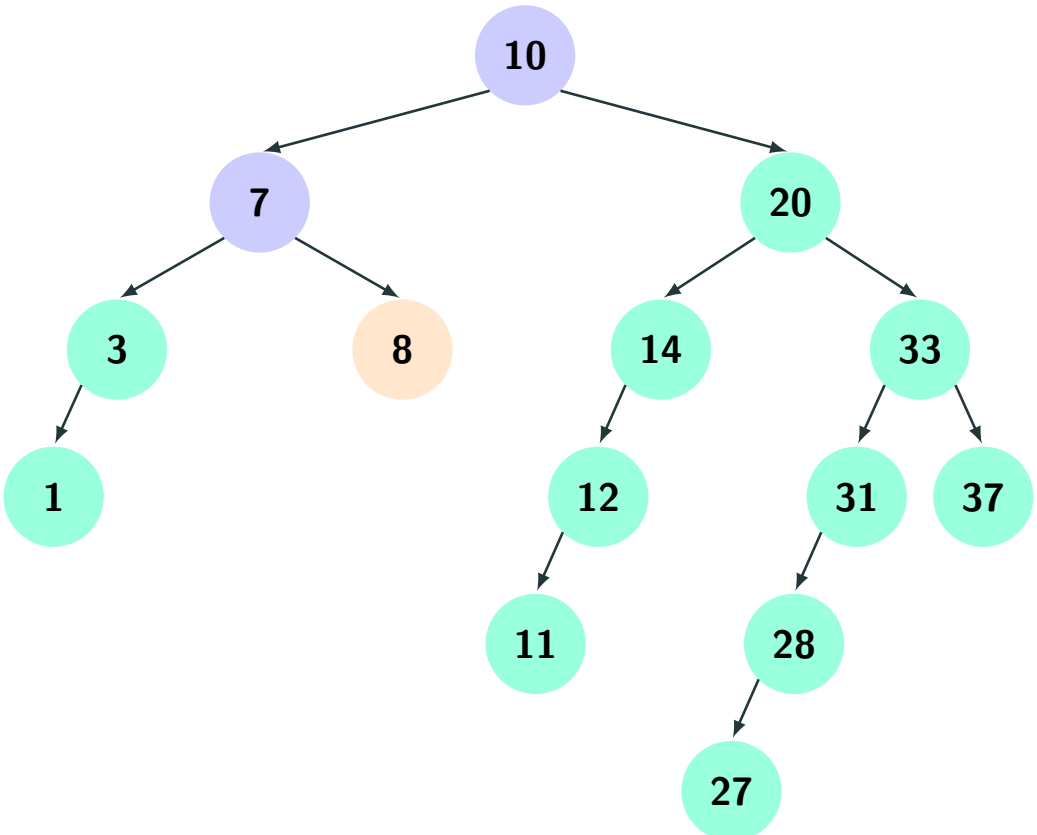
If the key to be deleted is in a leaf, just delete it. E.g., key 5:



Leaf node with key 5 deleted.

Deleting a key

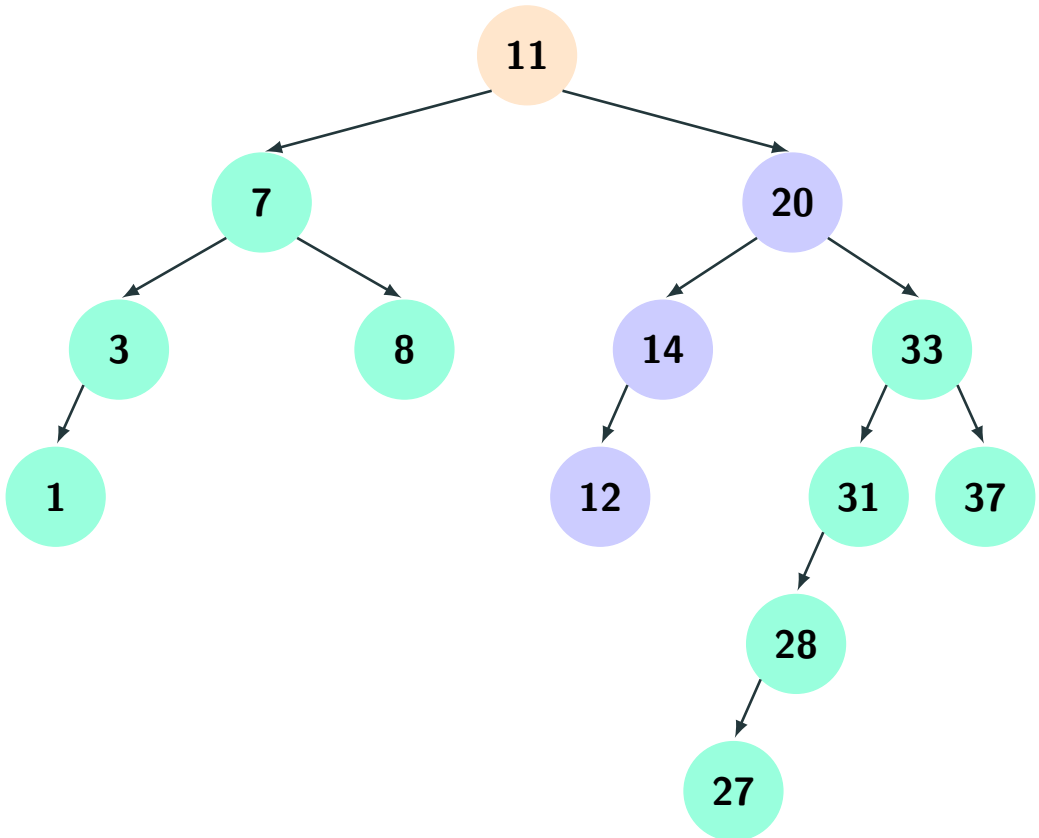
If the key to be deleted has one child, promote the child. E.g., key 9:



Key 9 deleted and key 8 promoted.

Deleting a key

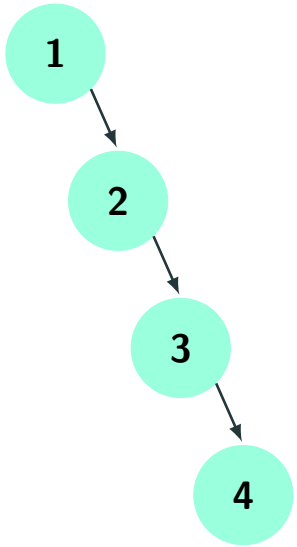
Otherwise, swap with successor or predecessor. E.g., key 10:



Key 10 swapped with its successor 11 and deleted.

Time complexity

- A BST is not a balanced tree and, in the worst case, may degenerate to a linked list. For example, if the keys are inserted in sorted order.



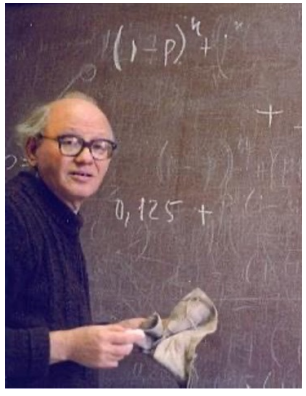
- **In the worst-case search, insertion and deletion are $\Theta(n)$.**

- Can we improve by keeping the tree balanced?

AVL Trees

AVL Trees

Adelson-Velskii Landis (AVL) tree is a height-balanced BST.



Georgy Adelson-Velskii

Evgenii Landis

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

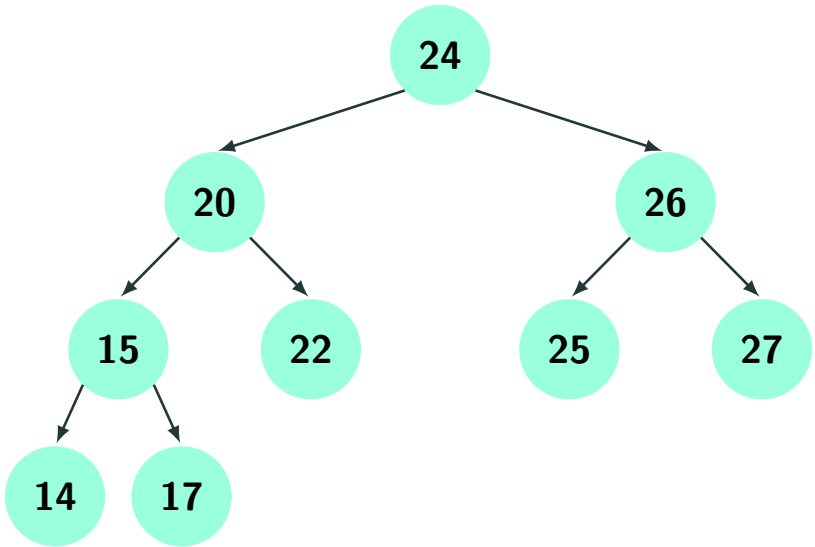
- If this property is violated at any time, rebalance immediately to restore it.

AVL Tree example

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

Is the following tree balanced according to the above definition?



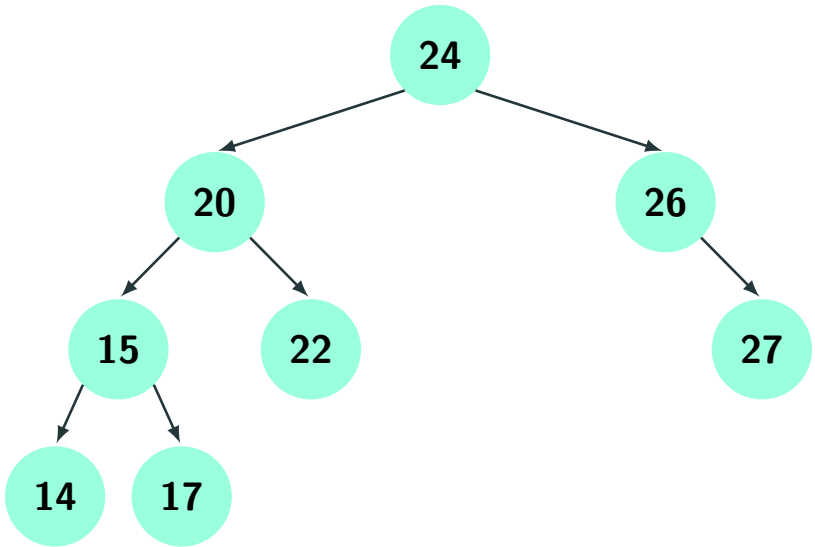
Yes.

AVL Tree example

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

Is it still balanced after deleting node 25?



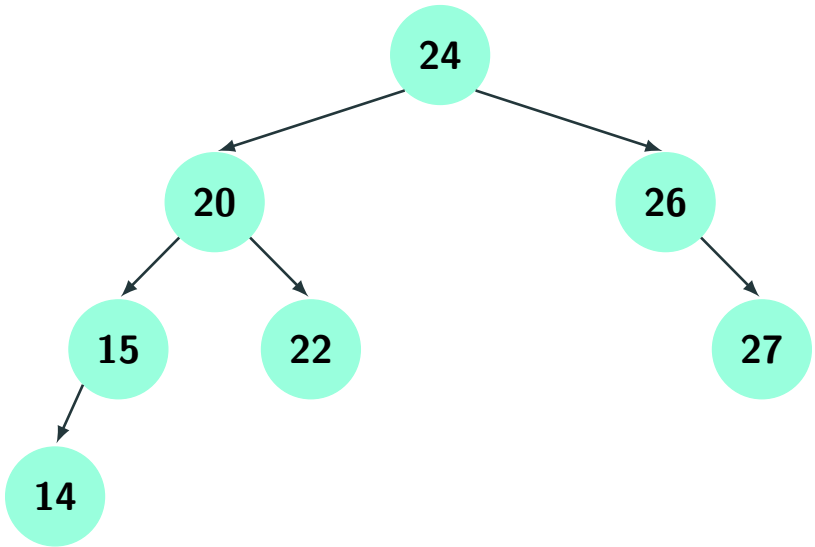
Yes.

AVL Tree example

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

What about after deleting node 17?



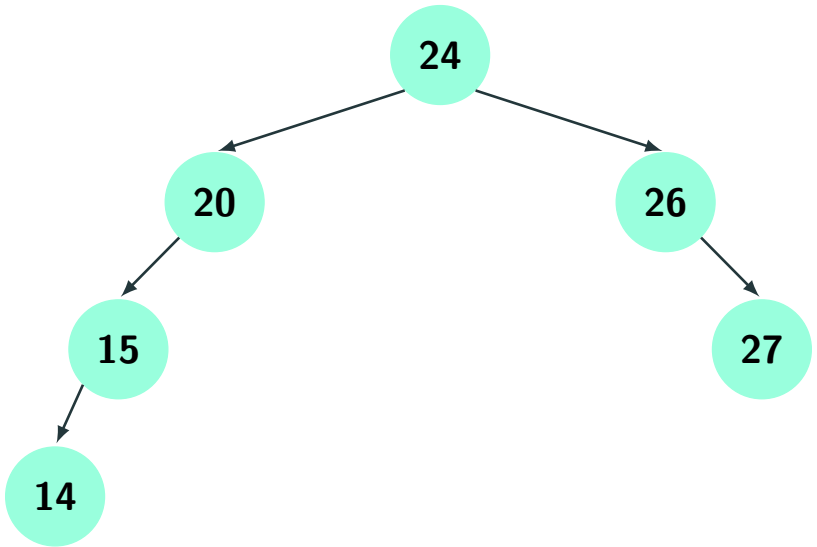
Quiz time!

AVL Tree example

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

After deleting node 22, is it still balanced?

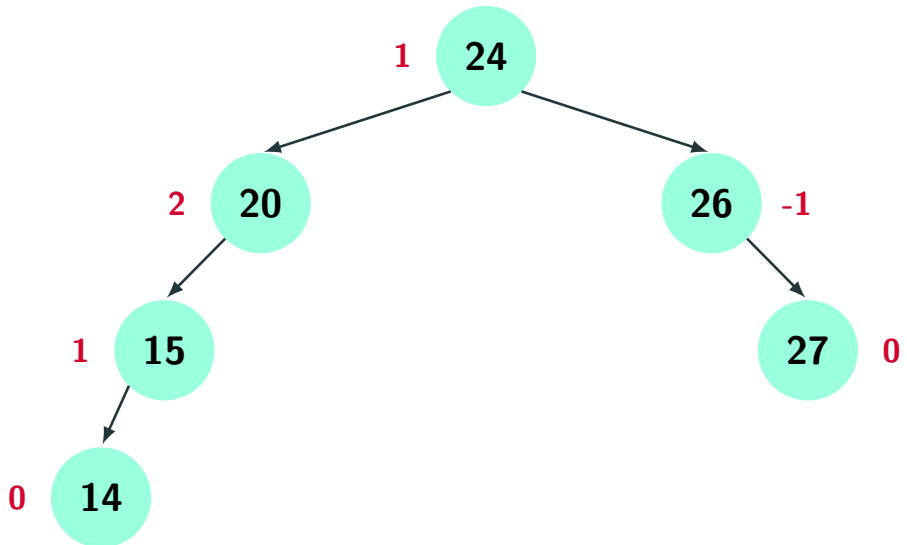


Balance factor

Balance of AVL Tree

For every node, heights of left and right subtrees differ by at most one.

Node's balance factor = height of left subtree − height of right subtree.

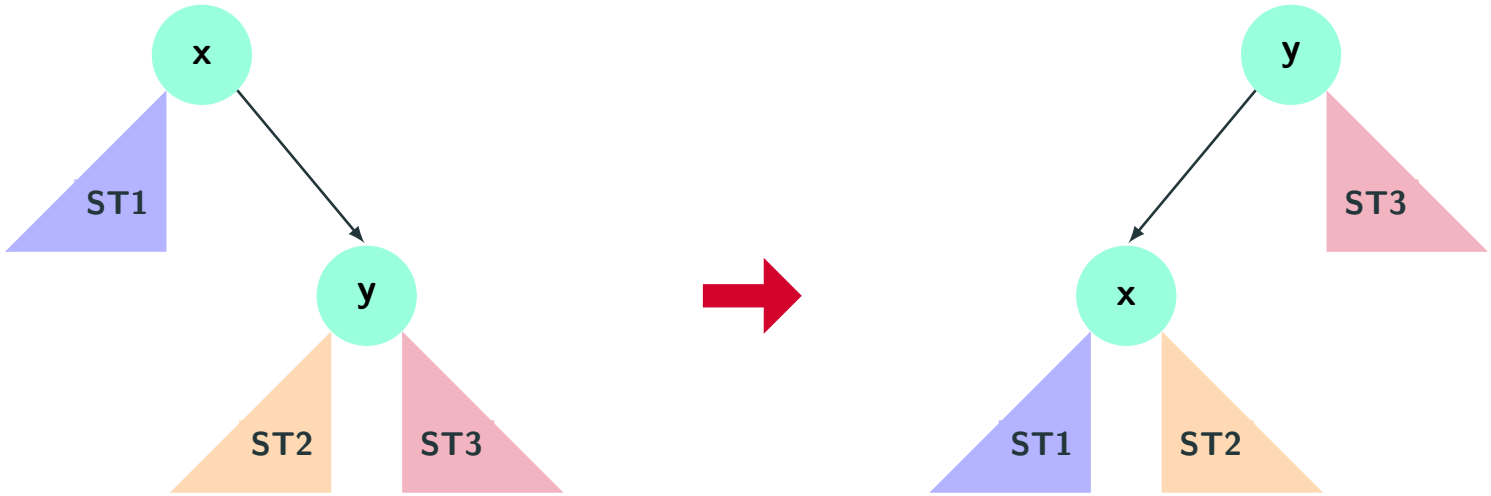


Imbalanced. Note that insertions can also cause imbalance.

Rebalancing AVL Trees

- Rebalancing is done from bottom up in any node that is imbalanced.
- There are four distinct cases:
  - **Left-left case:** node's balance factor is 2 and its left child has balance factor 0 or 1.
  - **Left-right case:** node's balance factor is 2 and its left child has balance factor -1.
  - **Right-right case:** node's balance factor is -2 and its right child has balance factor 0 or -1.
  - **Right-left case:** node's balance factor is -2 and its right child has balance factor 1.
- Rebalancing uses left and right rotations.

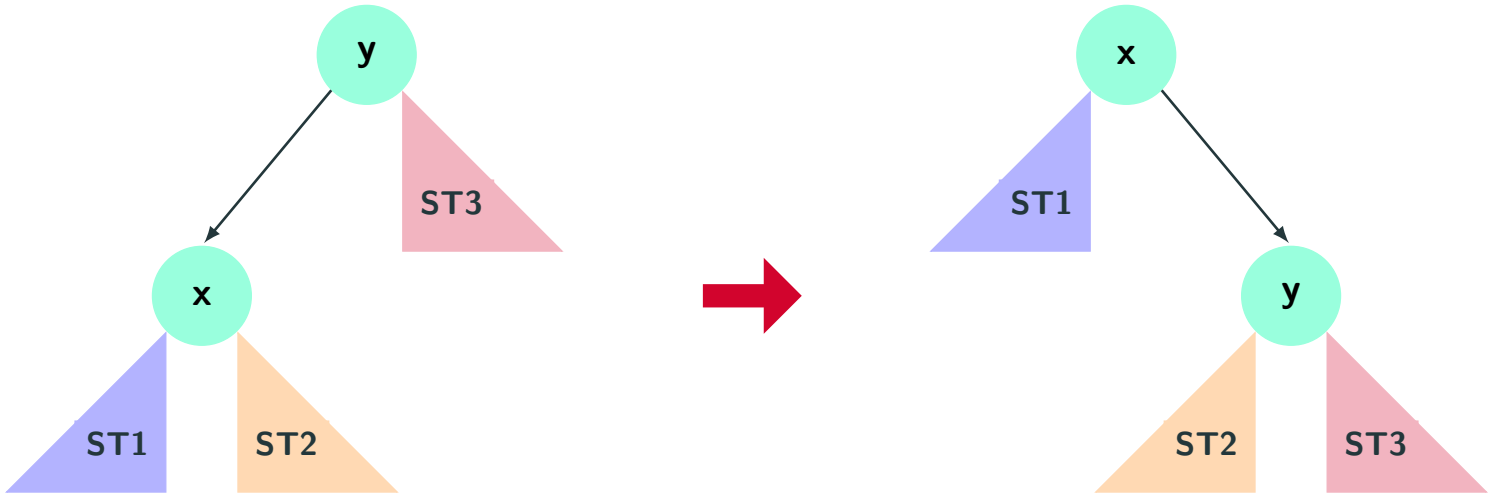
Left rotation



A left rotation of node x:

- Node y is promoted, and node x becomes its left child.
- The left subtree of node x stays that way.
- The right subtree of node y stays that way.
- The left subtree of node y becomes the right subtree of node x.

Right rotation

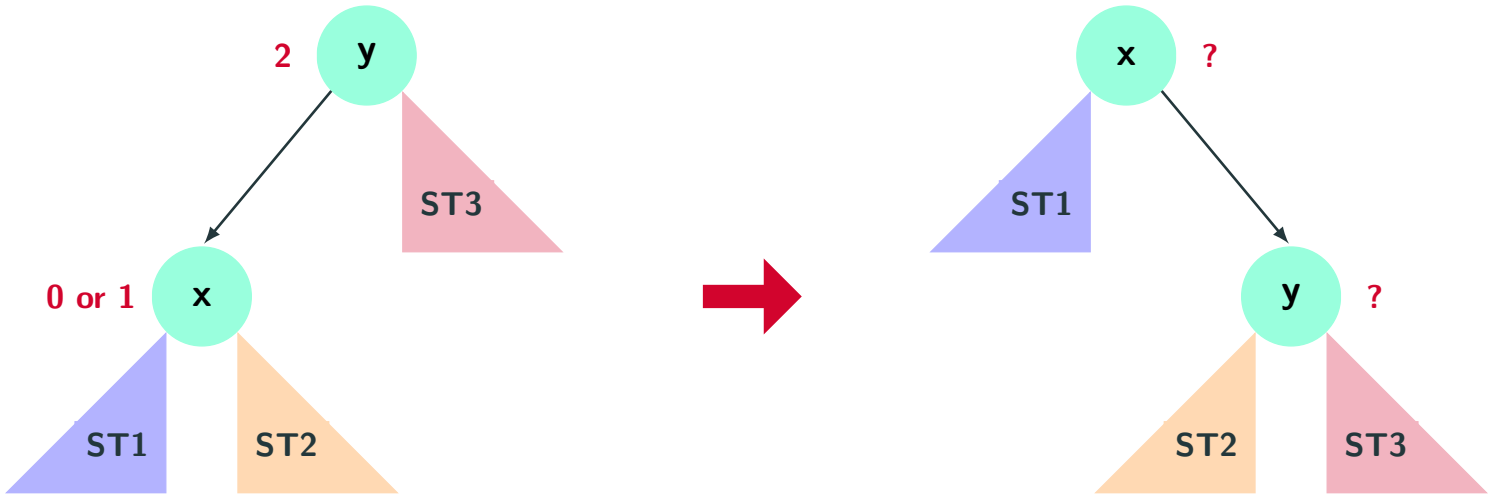


A right rotation of node y:

- Node x is promoted, and node y becomes its right child.
- The left subtree of node x stays that way.
- The right subtree of node y stays that way.
- The right subtree of node x becomes the left subtree of node y.

Handling a left-left case

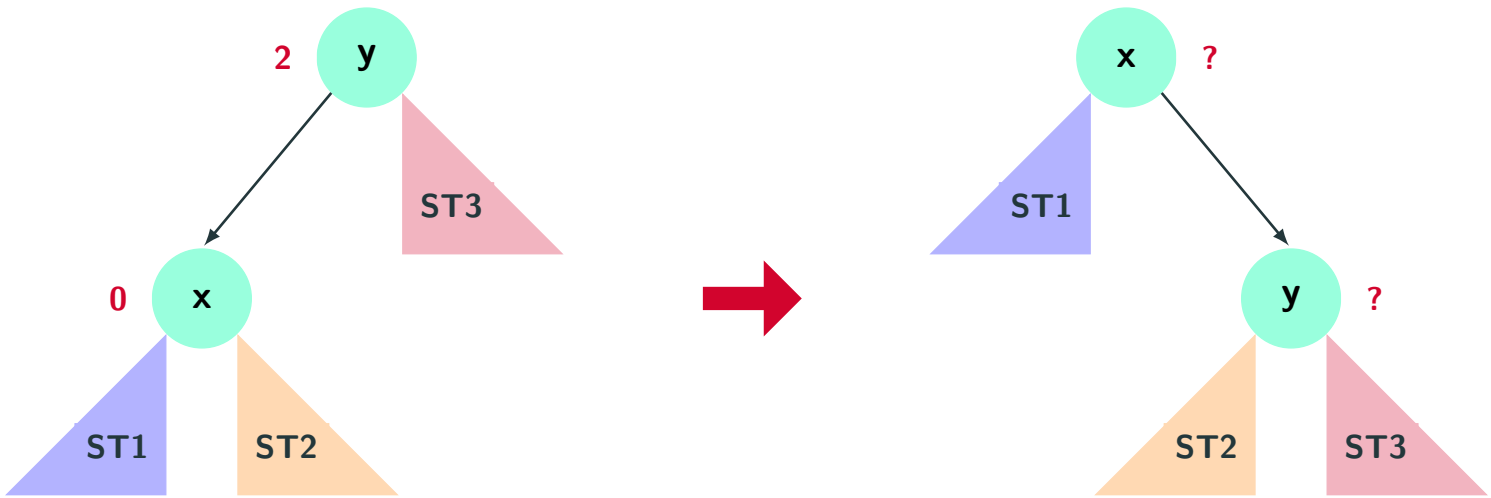
In the left-left case, we fix the imbalance by just doing a right rotation of the imbalanced node:



What are the new balance factors of x and y?

Handling a left-left case

Let's assume that the initial balance of x was 0 and let's denote height(**ST1**) by h:

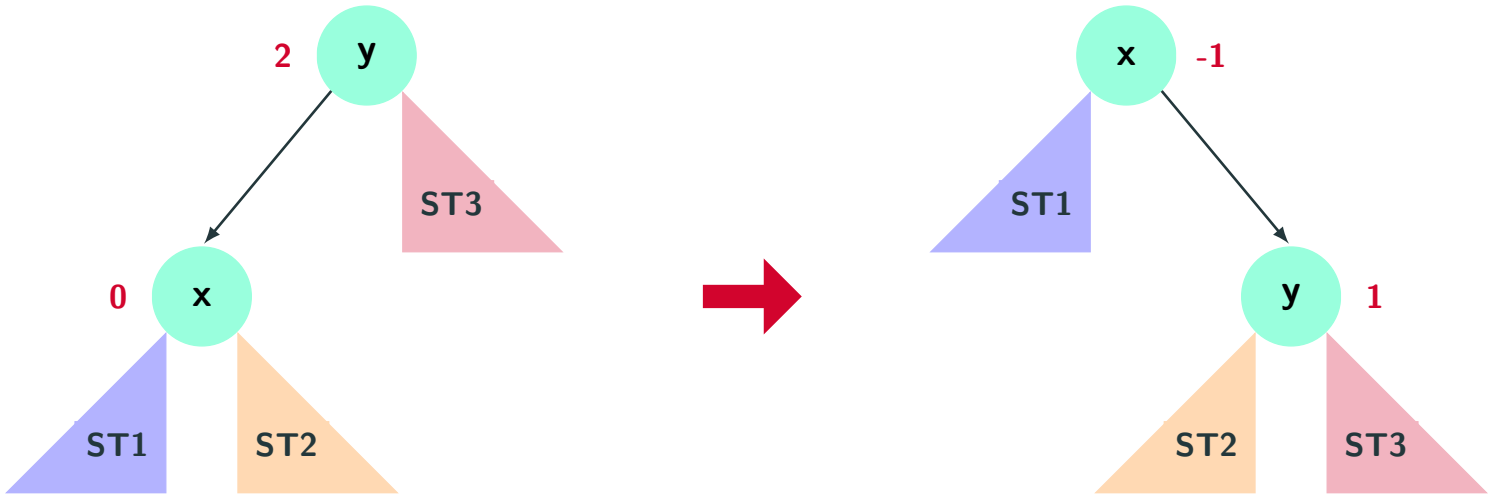


What are the values of height(**ST2**) and height(**ST3**)?

Quiz time!

Handling a left-left case

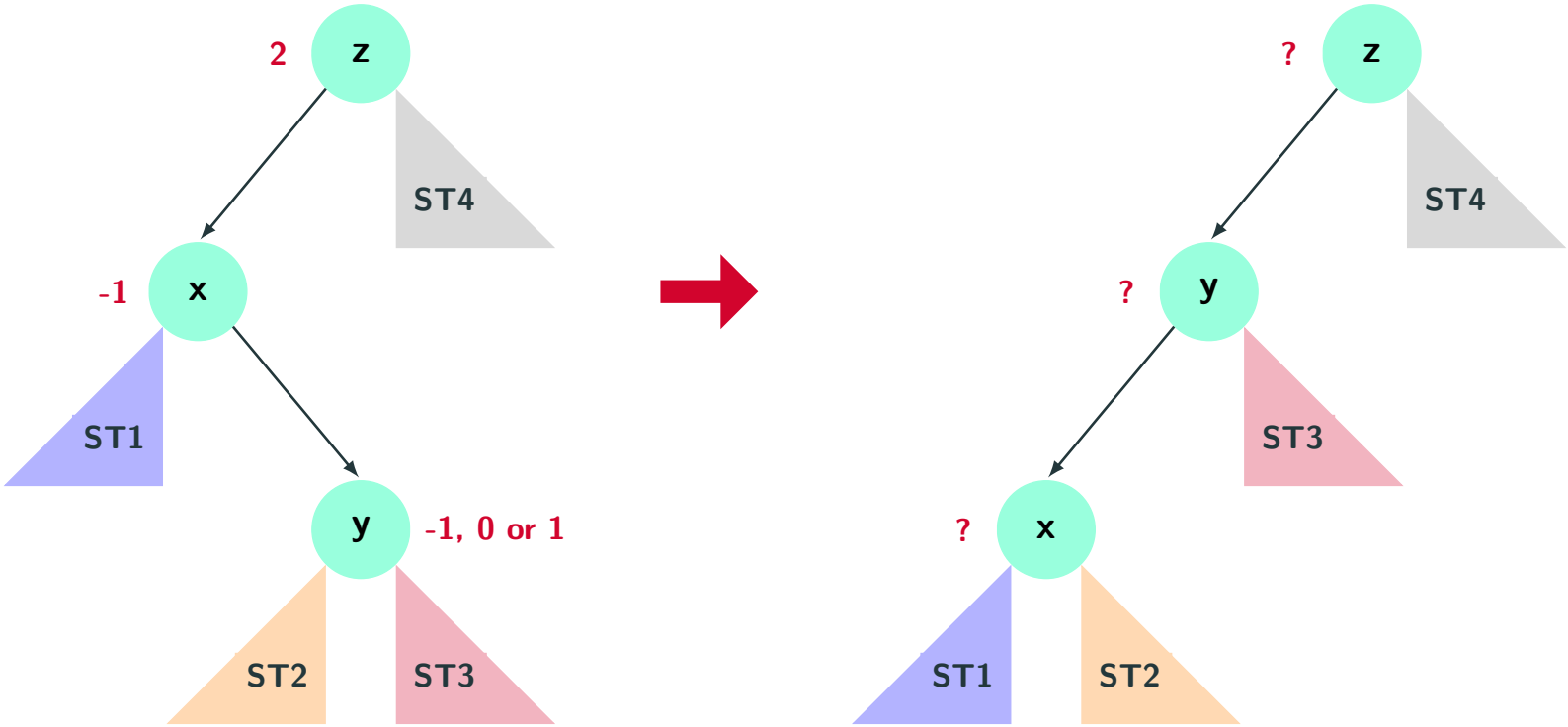
By analysing the initial balance factors, we conclude that height(**ST2**) $= h$, while height(**ST3**) $= h - 1$. It follows that:



If the initial balance of x is 1, the analysis is similar.

Handling a left-right case

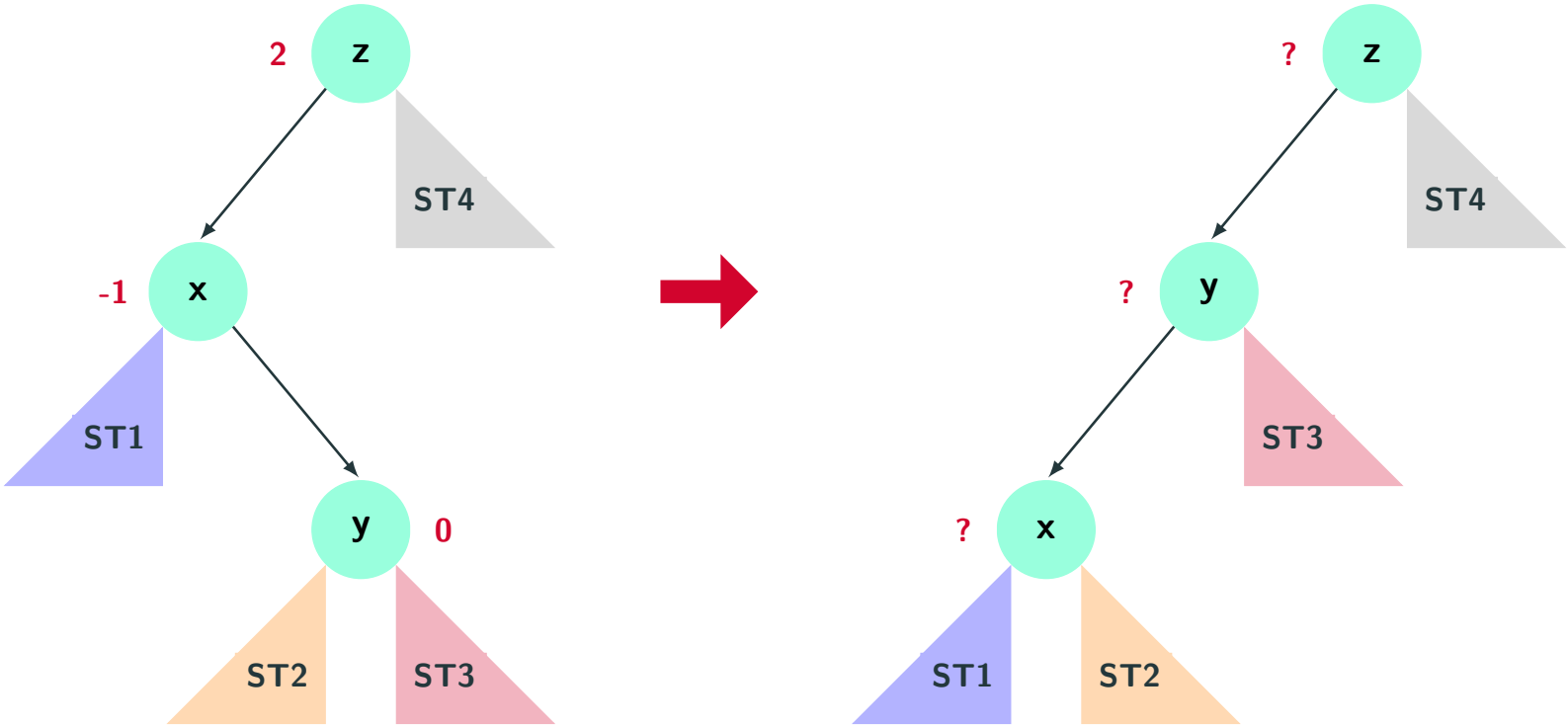
In the left-right case, we perform a left rotation on the left child in order to turn it into a left-left case:



What are the new balance factors of x, y and z?

Handling a left-right case

Let's assume that the initial balance of y was 0 and let's denote height(**ST2**) by h:

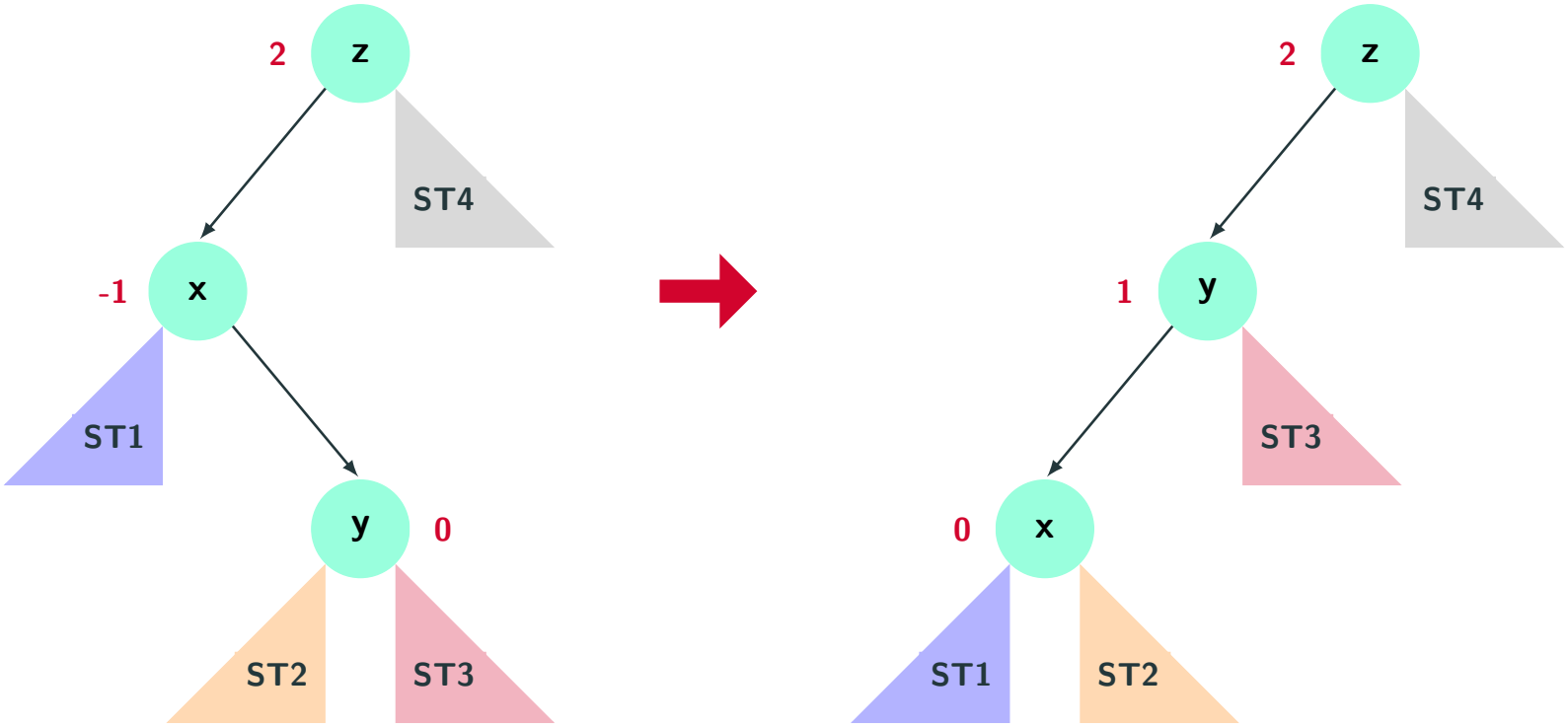


What are the values of height(**ST1**), height(**ST3**) and height(**ST4**)?

Quiz time!

Handling a left-right case

By analysing the initial balance factors, we can conclude that all subtrees' heights are h. It follows that:



Proceed with left-left case. Similar analysis for other initial balances of y.

A few remarks

- Right-right and right-left cases are mirror images of the left-left and left-right cases, and left as exercise.
- How to quickly updated the balance factors?
  - If storing balances directly, they can be cumbersome to update.
  - Store heights at each node instead as they are very easy to update.
  - Compute balances as needed by taking the height of the left subtree minus the height of the right subtree.

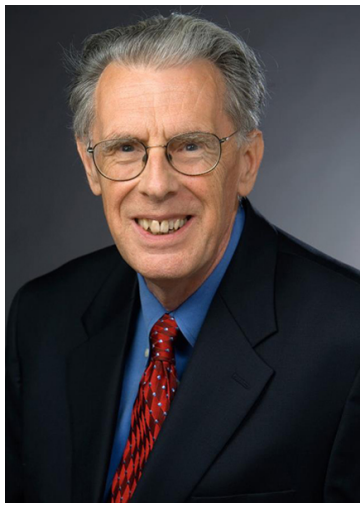
Time complexity of AVL Trees

- **Height:** $\Theta(\log n)$ because the tree is balanced (a proof of this is examined in next week's applied class sheet).
- **Search:** $\Theta(\log n)$ as it is a BST with height $\Theta(\log n)$.
- **Insertion/Deletion:** consists of the insertion/deletion plus the rebalancing cost.
  - AVL Tree is balanced before insertion/deletion.
  - Insertion/deletion of the element would follow BST procedures and therefore take time $\Theta(\log n)$.
  - An insertion/deletion can affect balance factor of at most $\Theta(\log n)$ nodes.
  - Balancing is bottom-up and performs 1 or 2 rotations for each node that is imbalanced, so constant time per imbalanced node.
  - Putting it all together, the overall time complexity is $\Theta(\log n)$.

2-3 Search Trees

Origins

2-3 Searches Trees were invented by John Hopcroft in 1970.



John Hopcroft (Turing Award 1986)

Perfectly Balanced 2-3 Search Trees

- When there are many dynamic insertions/deletions, it is quite expensive to maintain a perfectly balanced Binary Search Tree.
- A Perfectly Balanced 2-3 Search Tree (for short, 2-3 Search Tree) efficiently maintains a perfect balance in terms of the leaves' depth.
- It gives up on being binary and instead allows two different types of nodes: 2-Nodes and 3-Nodes.

Types of Nodes

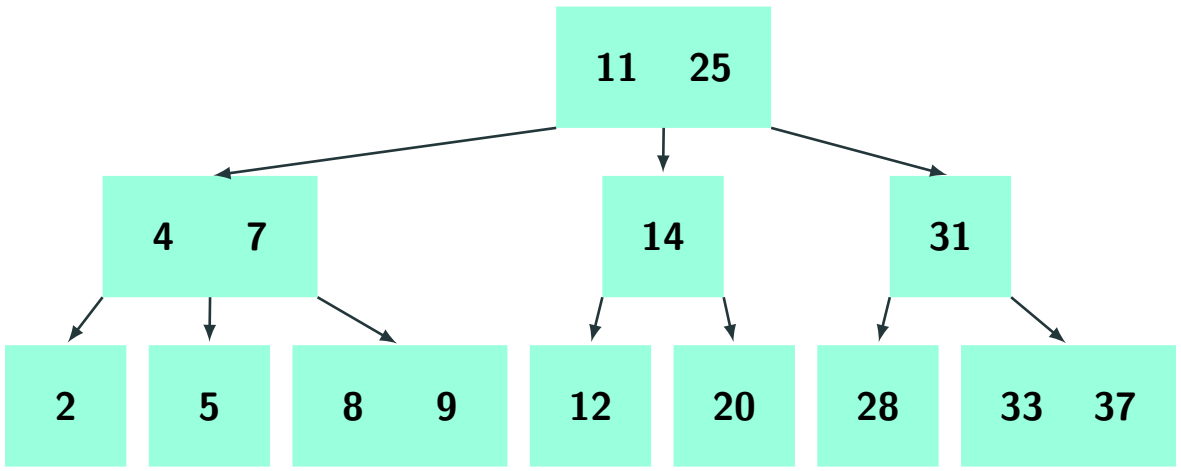
2-Node

- Similar to BST nodes
- 1 key K_1
- 2 children
- Left subtree only contains keys $K < K_1$
- Right subtree only contains keys $K > K_1$

3-Node

- 2 keys K_1, K_2 with $K_1 < K_2$
- 3 children
- Left subtree only contains keys $K < K_1$
- Middle subtree only contains keys $K_1 < K < K_2$
- Right subtree only contains keys $K > K_2$

2-3 Search Tree example

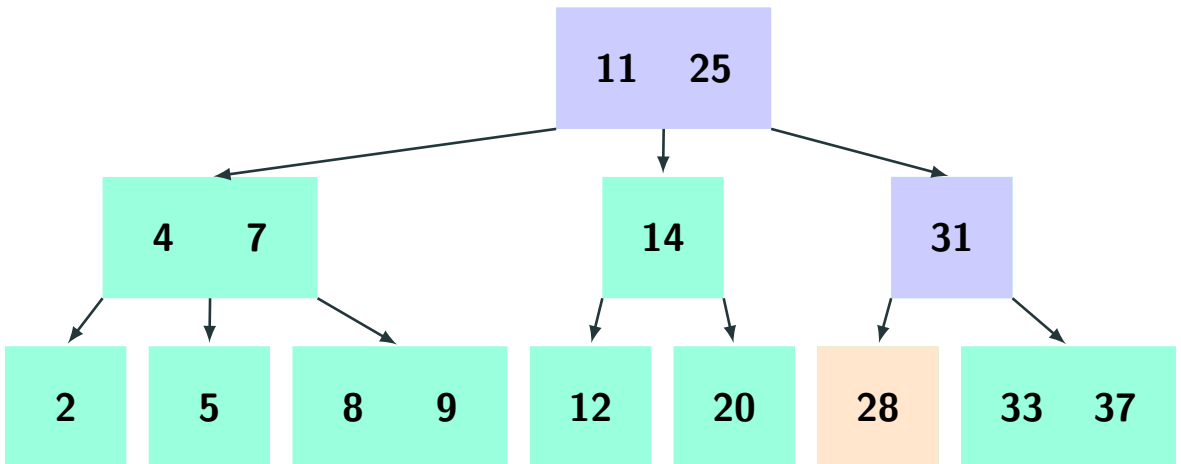


Operations in 2-3 Search Trees

- **Searching:** straightforward generalisation of BST's recursive procedure.
- **Insertion:** start by performing a search for the key. If not founded:
  - If the leaf is a 2-Node, just turn it into a 3-Node and add key to it.
  - If the leaf is a 3-Node, temporarily turn it into a 4-Node and add key to it.
  - A 4-Node is not a valid part of a 2-3 Search Tree, so adjustments will be needed to get rid of the 4-Node.

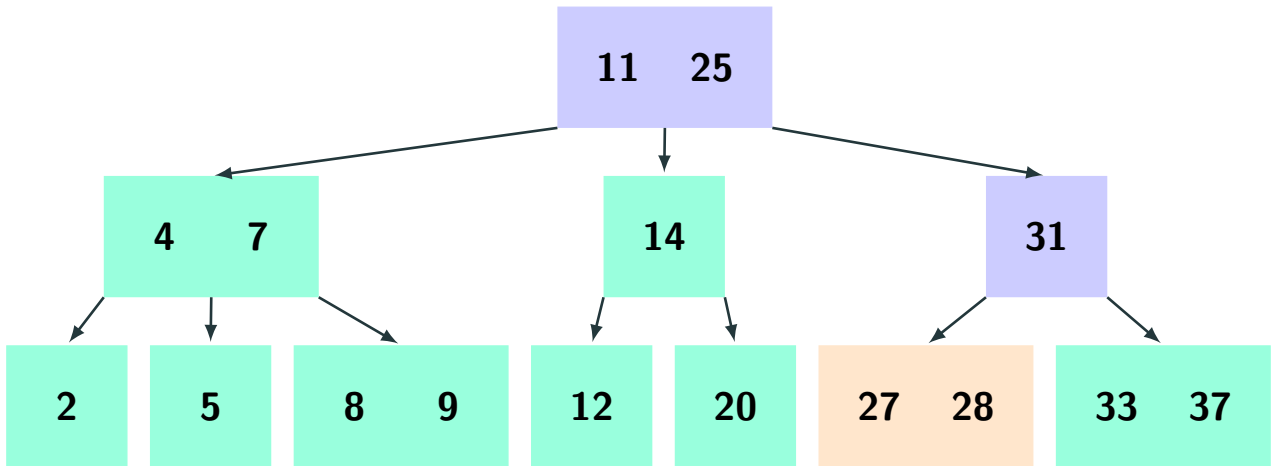
Let's add some key...

Let's see what happens when we add key 27 to the previous example:



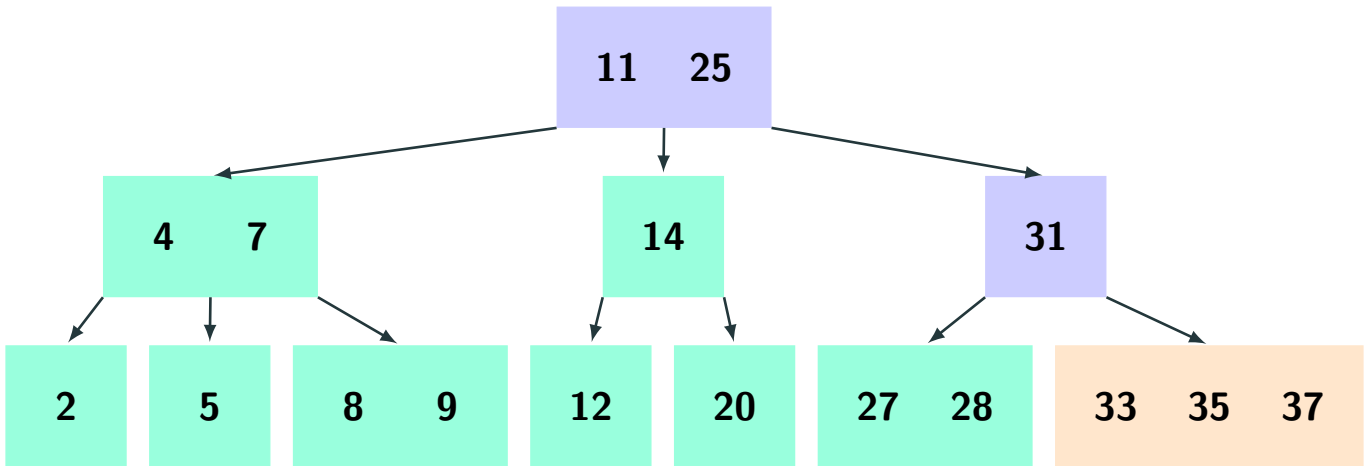
Let's add some key...

As the leaf was a 2-Node, just turn it into a 3-Node and add key 27 to it:



Another insertion

We now add key 35:

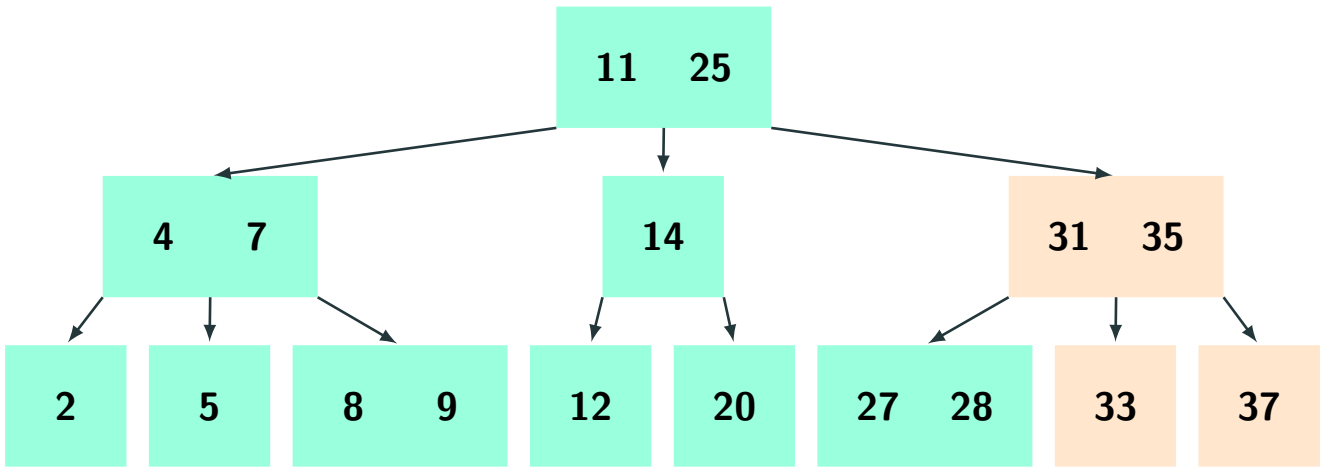


Problem: We got a 4-Node.

Solution: Split it into two 2-Node and promote middle key to the parent.

Another insertion

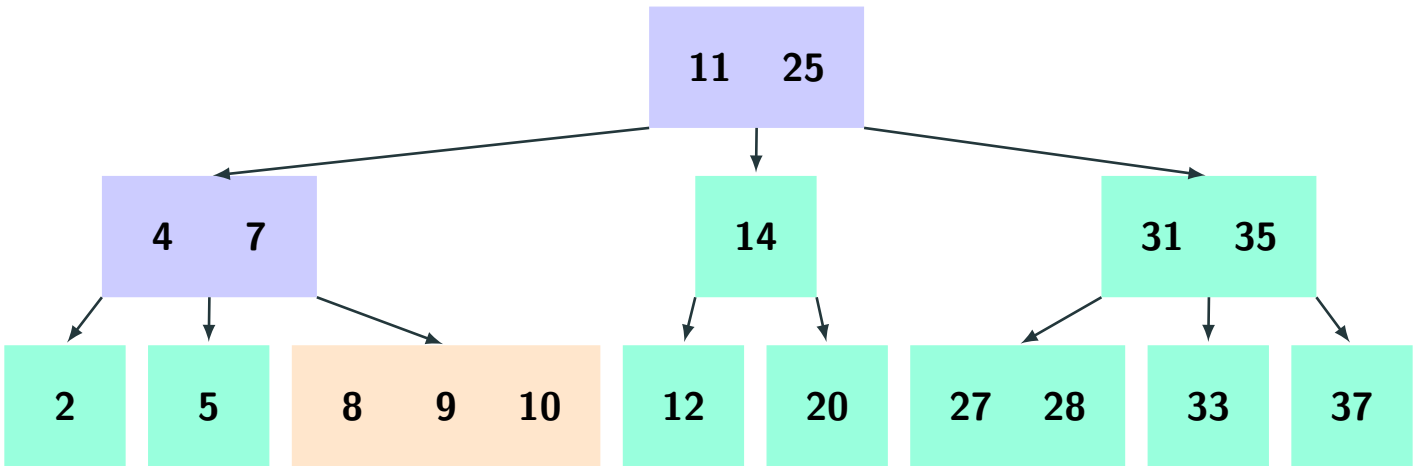
Doing the split and the promotion:



As the parent becomes a 3-Node, we are done with the adjustments. If the parent node turned into a 4-Node, we would need to proceed recursively upwards with the adjustments.

Adding a final key

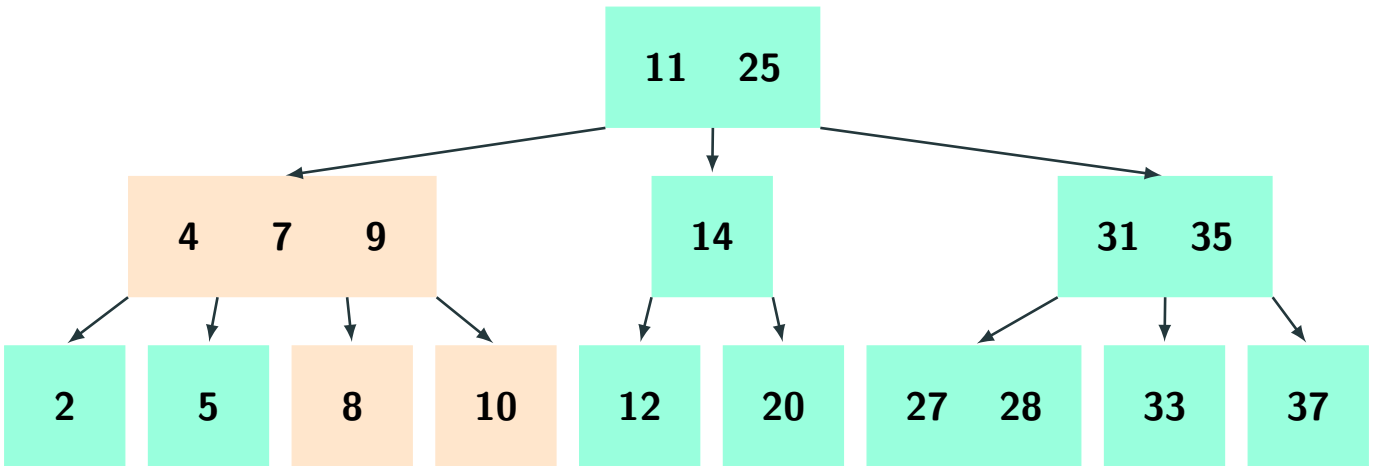
Adding key 10:



We need to fix the 4-Node.

Adding a final key

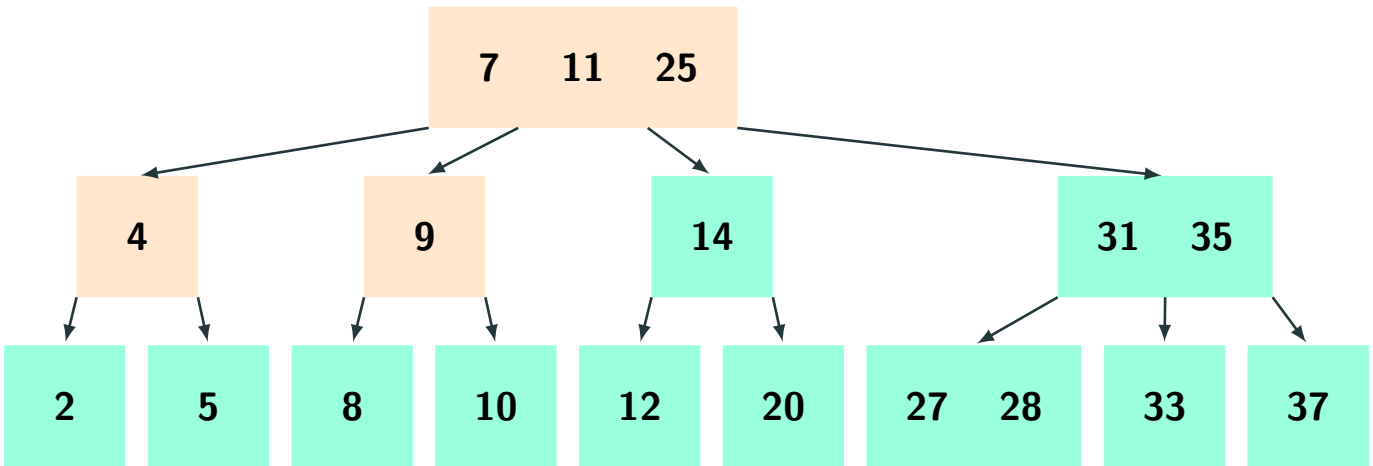
Doing the split and the promotion:



The parent turned into a 4-Node that needs to be fixed.

Adding a final key

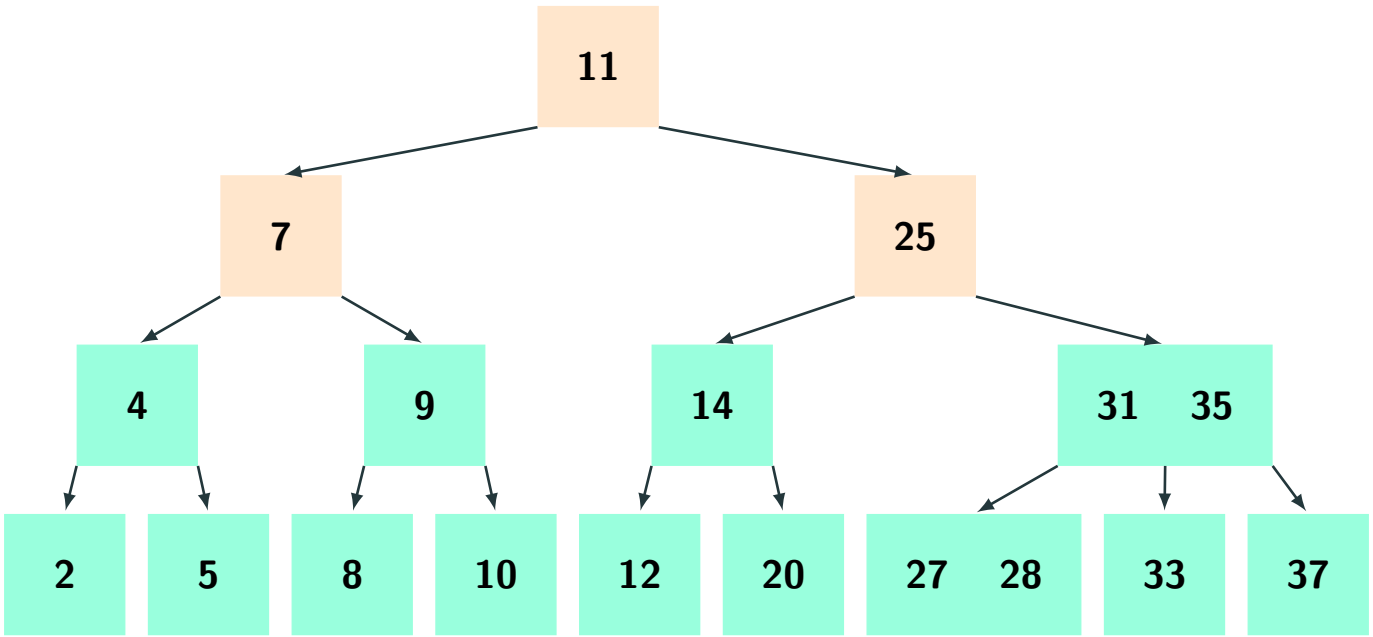
Doing the split and the promotion:



We got a 4-Node at the root. We fix that by turning it into three 2-Nodes and creating a new root level.

Adding a final key

Final result:



A few remarks

- In BSTs the number of nodes always grow by 1 when a new key is inserted as a leaf node. In contrast, in 2-3 Search Trees no leaf node is directly added when a key is inserted.

- Some insertions will create no new nodes at all, while other insertions will trigger a series of adjustments that might result in the creation of up to $\Theta(\log(n))$ new nodes (at most one new node per level).

- Some nodes change type during the insertion and adjustments.

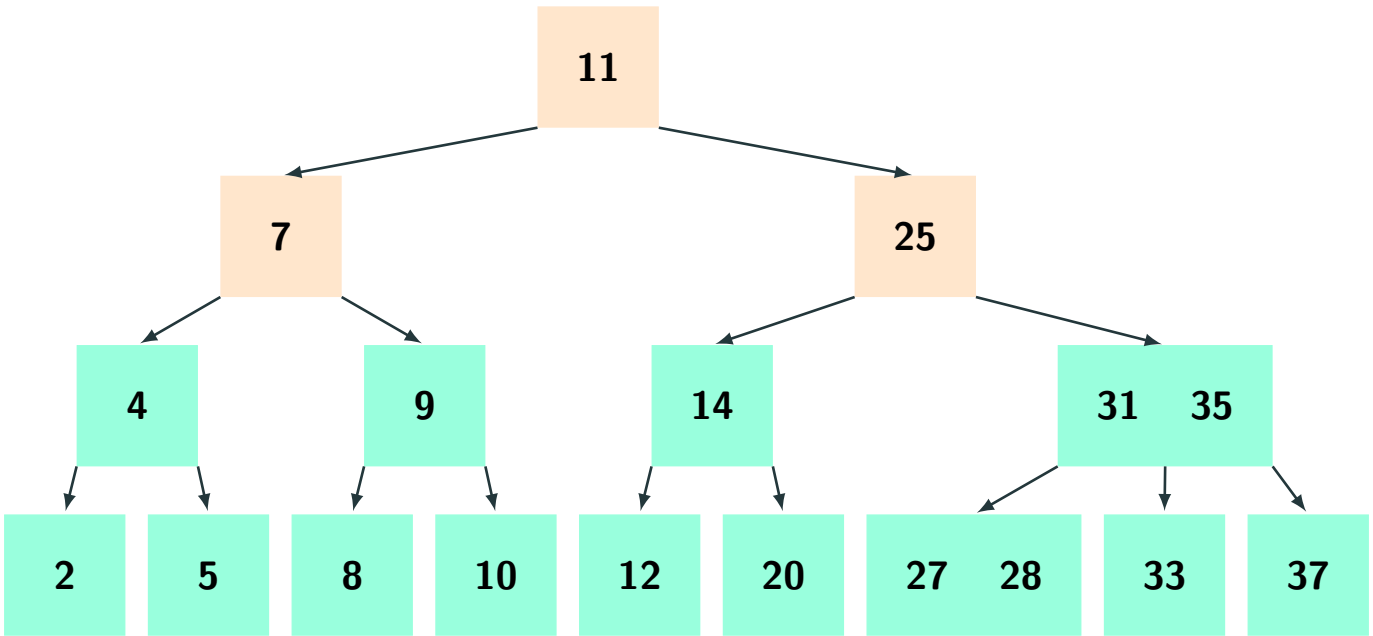
- No new level is ever added at the bottom of the tree. Sometimes the adjustments will result in the creation of a new root node.

Deletion

- Let's start with the case in which the key to be deleted is already in a leaf node.
- Even in this case, there would be a balancing issue if the leaf is a 2-Node.
- **Main idea:** make necessary adjustments on our way down to keep the invariant that we are always moving to 3-Nodes or 4-Nodes, but never to 2-Nodes.
- When we get to the leaf node we can simply delete the key as it is not a 2-Node.
- Traverse upwards eliminating any temporary 4-Nodes that remained.

Deleting the minimum key

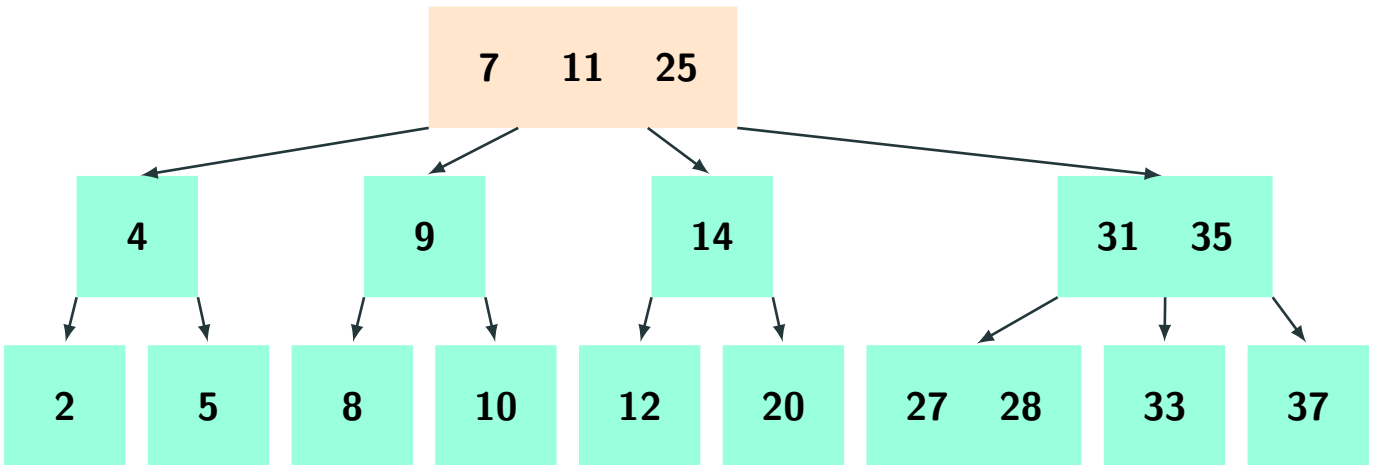
Let's delete the smallest key of the previous example:



At the start, the root and both of its children are 2-Nodes.

Deleting the minimum key

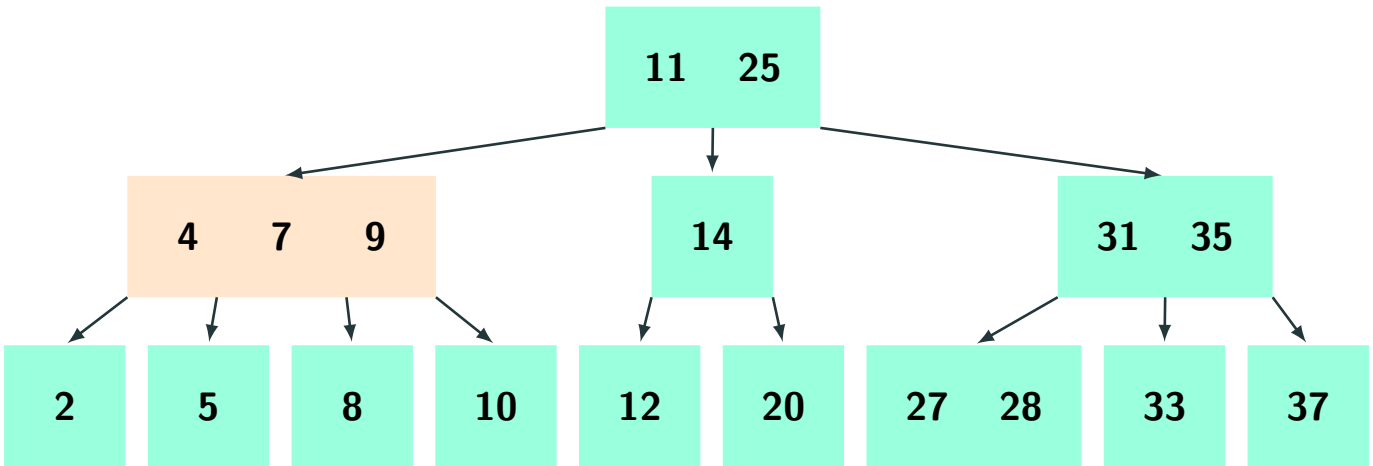
Merge them into a 4-Node:



The two left most children of the current node are 2-Nodes. We merge them and the smallest key of the current node to form a 4-Node, and proceed to that node.

Deleting the minimum key

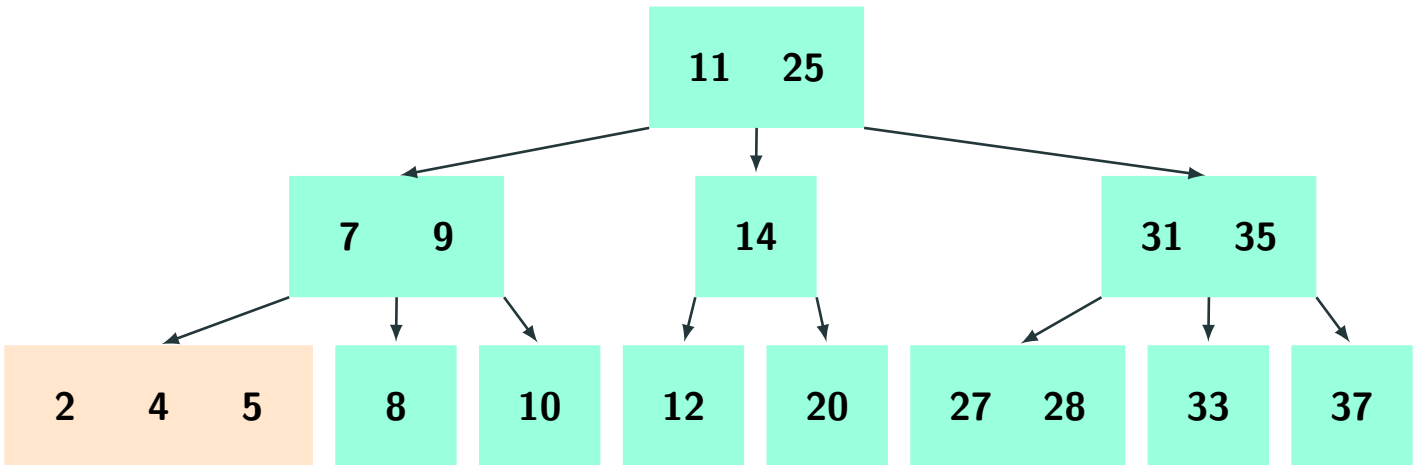
Getting the new 4-Node and moving into it:



Again, the two left most children of the current node are 2-Nodes, so repeat the same steps.

Deleting the minimum key

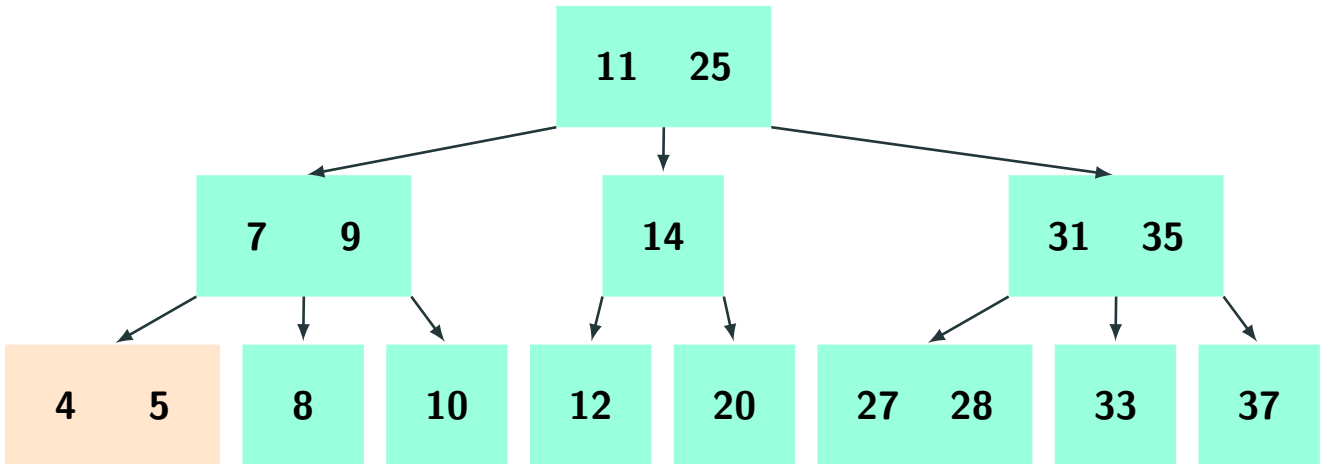
Creating the new 4-Node leaf:



Now we can simply delete key 2.

Deleting the minimum key

Final result:



There is no 4-Node to undo on the way up, so we are done with the deletion.

A few remarks

- If the key to be deleted is in an internal node, proceed downwards to find that key and continue downwards to find its successor while always keeping the same invariant. Swap the key with its successor, and delete the key from the leaf node.

- More details in the course notes.

- For full details about the deletion procedure, please check Chapter 3 of the 4th edition of *Algorithms* by Robert Sedgewick and Kevin Wayne.

Height of the tree

- All leaves are in the same level.
- If the height is h, an easy lower bound on the number of nodes m is given by $m \geq 2^{h+1} - 1$ (as that would be the number of nodes if there were only 2-Nodes).
- We get that $h = O(\log(m))$.

Time complexity

- Search operation only visit one node per level on its way down.
- In the insertion operation the adjustment operations might visit up to 1 node per level on the way up.
- On each visited node the operations performed are $\Theta(1)$.
- Number of keys n is such that $n \geq m$.
- Putting all together, search and insertion operations are $O(\log(n))$.
- Deletion also only do a pass downwards and a pass upwards, doing constant amount of operations at each level, so it is also $O(\log(n))$.

Practical considerations

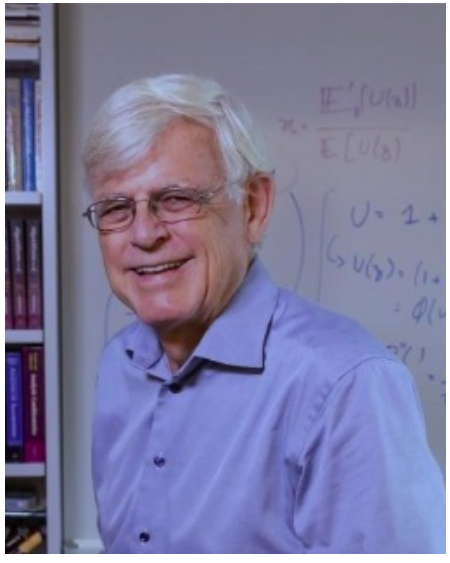
- 2-3 Search Trees and other B-Trees variants are really cool data structures.
- However, the fact that there are different types of nodes creates many different cases that need to be handled in an implementation.
- That makes it more cumbersome to implement and introduces overheads in terms of the concrete efficiency (i.e., the hidden constant in the Big-O notation).
- For those reasons often Red-Black Trees (which are BSTs related to B-Trees) are used in real-world deployments.

Red-Black Trees

Red-Black Trees

The first variant of Red-Black Trees (RBT) were introduced by Leonidas J. Guibas and Robert Sedgewick in 1978.

Leonidas Guibas



Robert Sedgewick

This unit focuses on the Left-Leaning RBT variant that was introduced by Sedgewick in 2008.

Left-Leaning Red-Black Tree

- Goal: simplifying the analysis and implementation of RBTs.
- There is a one-to-one correspondence between a Left-Leaning RBT and a Perfectly Balanced 2-3 Search Tree.
- For conciseness, we will most refer to it as Red-Black Tree or even RBT.

Left-Leaning Red-Black Tree

Left-Leaning Red-Black Tree

A Left-Leaning RBT is a BST coloured black or red such that:

1. The colour of a non-root node is determined by the colour of its single incoming edge. The root is by definition black.

2. All red links lean left, i.e., all red links go to a left child.

3. No node is adjacent to two red edges.

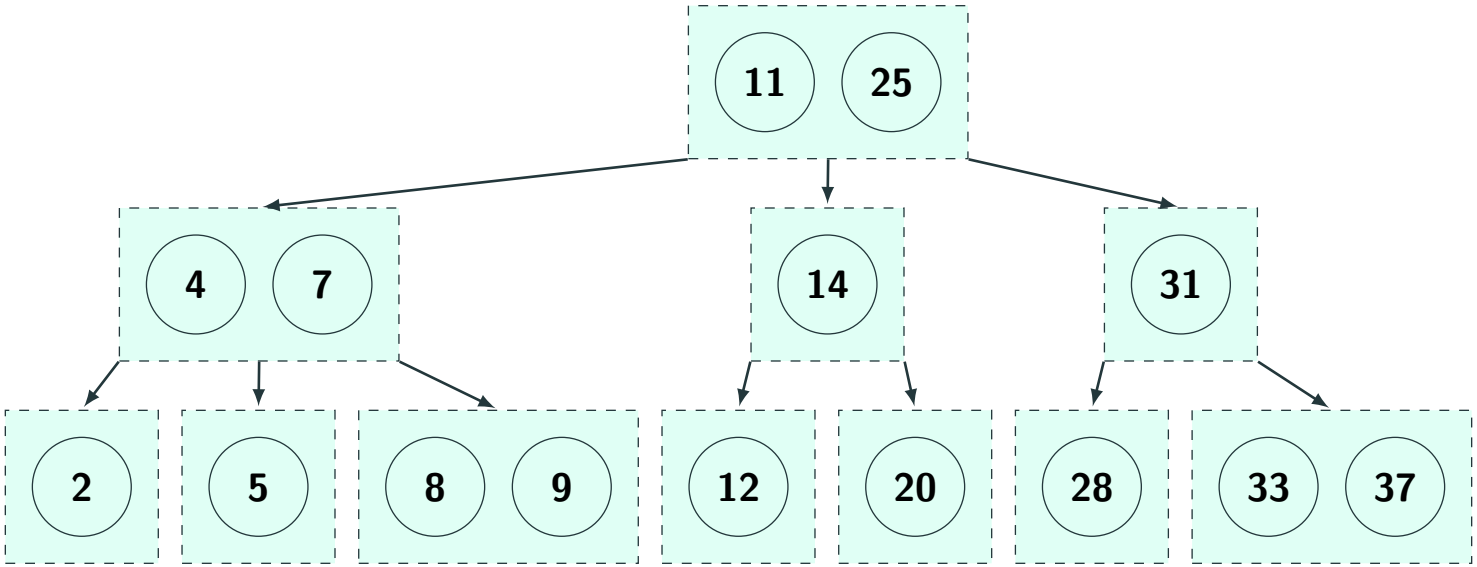
4. The tree has perfect black balance: the number of black edges between any node and each of its descendant leaves are the same.

A few more details about RBTs

- The number of black edges between the root node and any leaf is called the **black height** of the tree.
- The colour information is normally stored as a single bit inside the node (note that the edge's colour is implied by the node's colour and vice-versa).

Equivalence with 2-3 Search Trees

Start by separating the keys in 3-Nodes to become independent nodes:



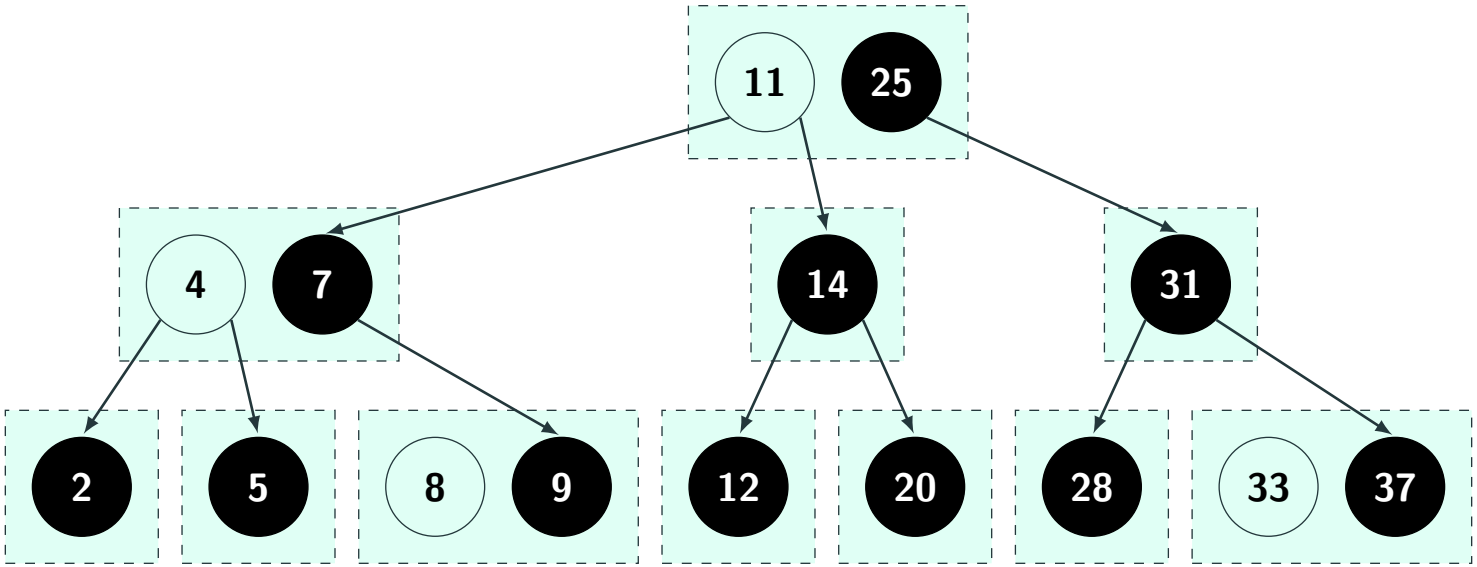
What happens with the existing edges?

All existing edges are coloured black and are redirected as follows:

- The incoming edge of an original 3-Node will now point to the right node after the separation.
- The left child of an original 3-Node will now become a left child of left node after the separation.
- The middle child of an original 3-Node will now become a right child of left node after the separation.
- The right child of an original 3-Node will now become a right child of right node after the separation.

Getting black nodes

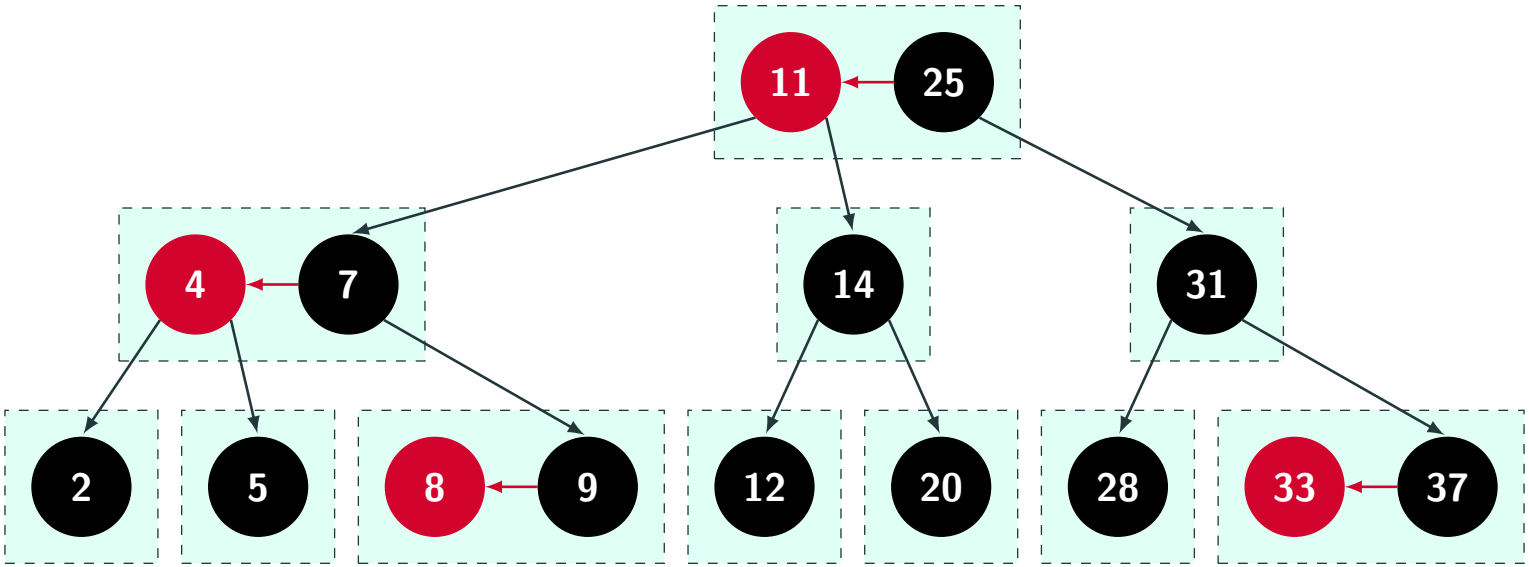
Redirecting the existing edges and painting black nodes accordingly:



Next, for each original 3-Node, add a red edge from the separate right node into the separate left node.

Getting red nodes

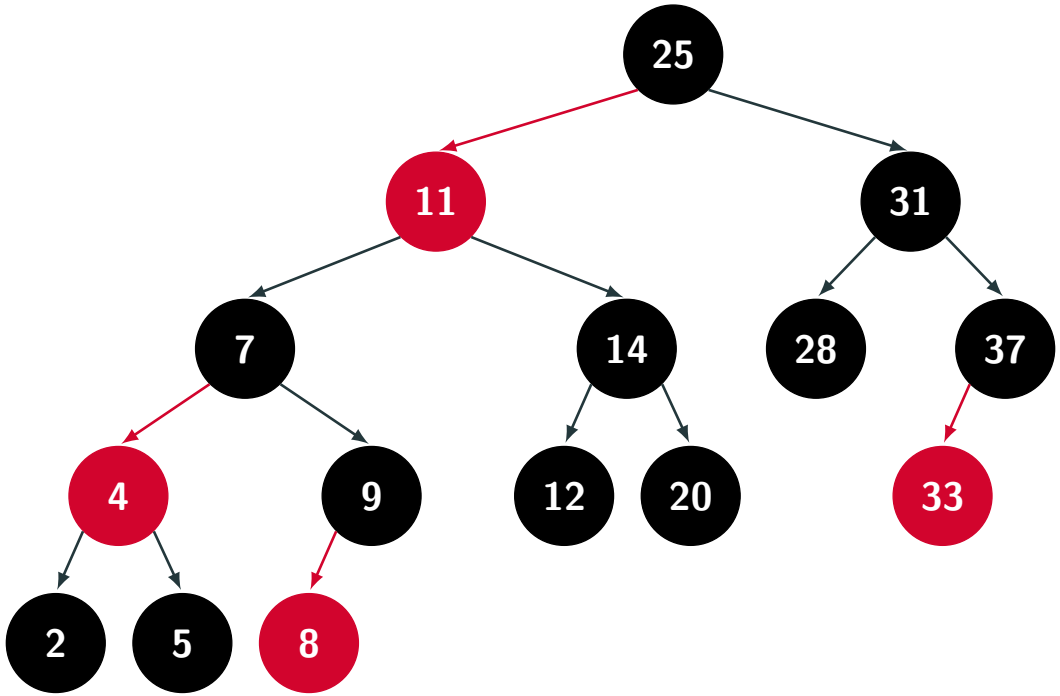
Adding the internal edges and painting red nodes accordingly:



Finally, just unflatten the tree so that all arrows go downwards.

Corresponding RBT

RBT that corresponds to the original 2-3 Search Tree:



We are essentially using the colouring information to encode 3-Nodes within the standard framework of BSTs.

What is the (black) height?

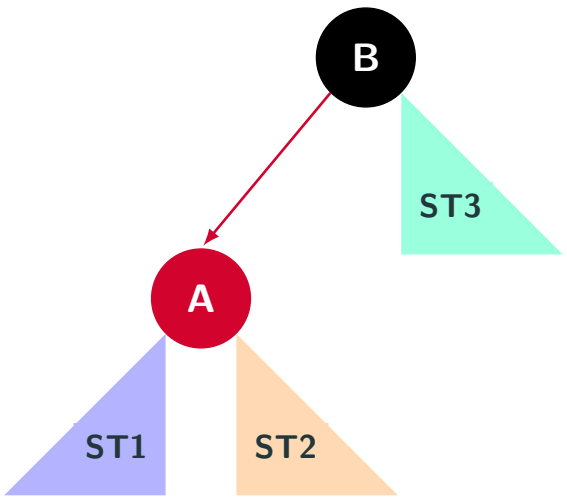
- **Black height** of the RBT is the same as the height of the original 2-3 Search Tree, so $O(\log(n))$ for a tree with n keys.
- Any path from the root to a leaf never has two consecutive red edges.
- So the **height** of the RBT is at most equal to twice its black height plus one, and thus also $O(\log(n))$.

Operations in RBT

- **Searching:** RBT is a BST, just ignore colours and do BST search.
- **Insertion:** series of steps that produce the same end result as if the insertion was done in the corresponding 2-3 Search Tree:
  - If the key is not found, insert it as a red node.
  - This might trigger adjustments on the way up from the point of insertion to the root in order to keep the RBT properties.
  - Inserting the new key as a red node preserves the perfect black balance of the RBT.

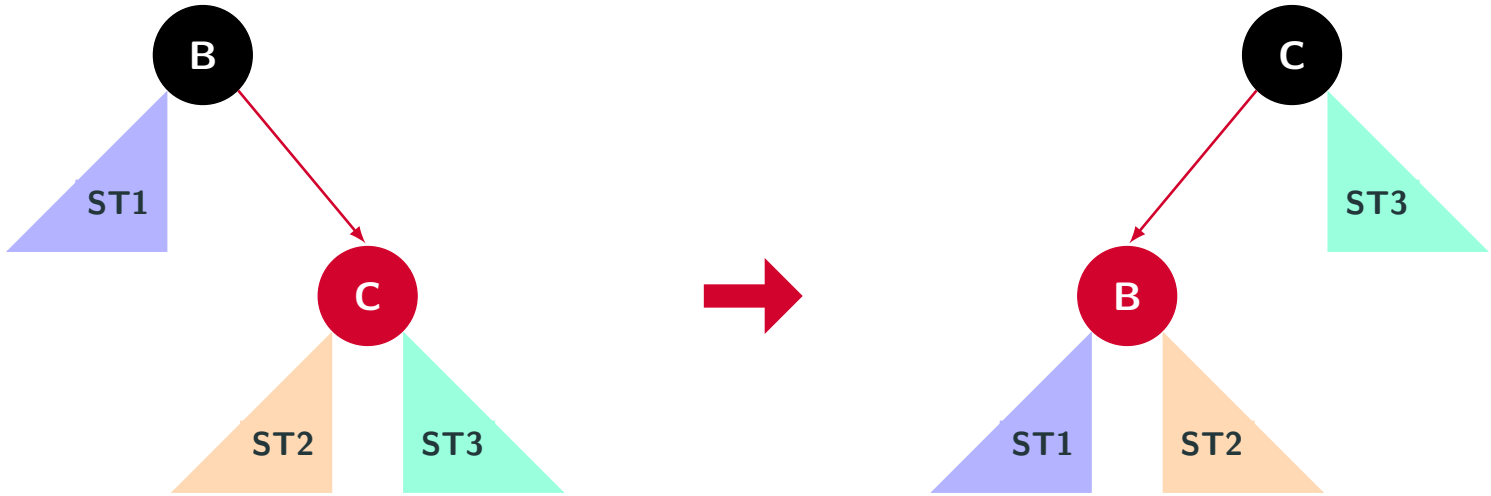
Insertion into a 2-Node

- Red node is being added as a child of a black node that previously had no red child.
- If it is being added as a left child, no changes are necessary:



Insertion into a 2-Node

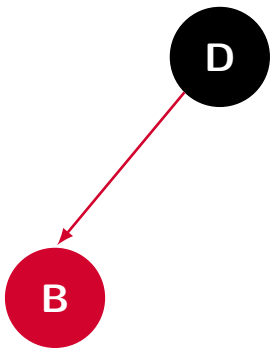
- Otherwise, it is being added as a right child and needs to be fixed by performing a left rotation:



- In all rotations the edges preserve the colour (but its direction is changed to keep pointing down) while the nodes switch colours appropriately to match the colours of their new incoming edges.

Insertion into a 3-Node

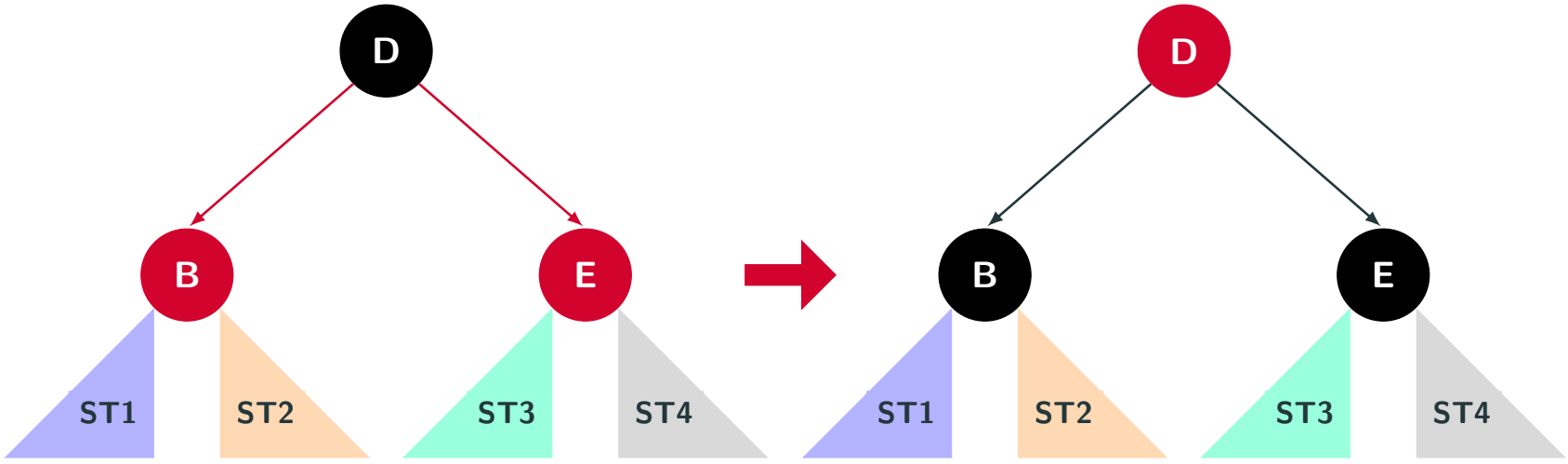
- But we also need to handle the case in which a red node is added as a child in a 3-Node. Let's consider that the 3-Node before the addition was:



- There are 3 cases depending on whether the new red key is smaller than both previous keys, bigger than both, or in between them.

1st case

- If the new red key is bigger than both previous keys this is the easiest case. For example:

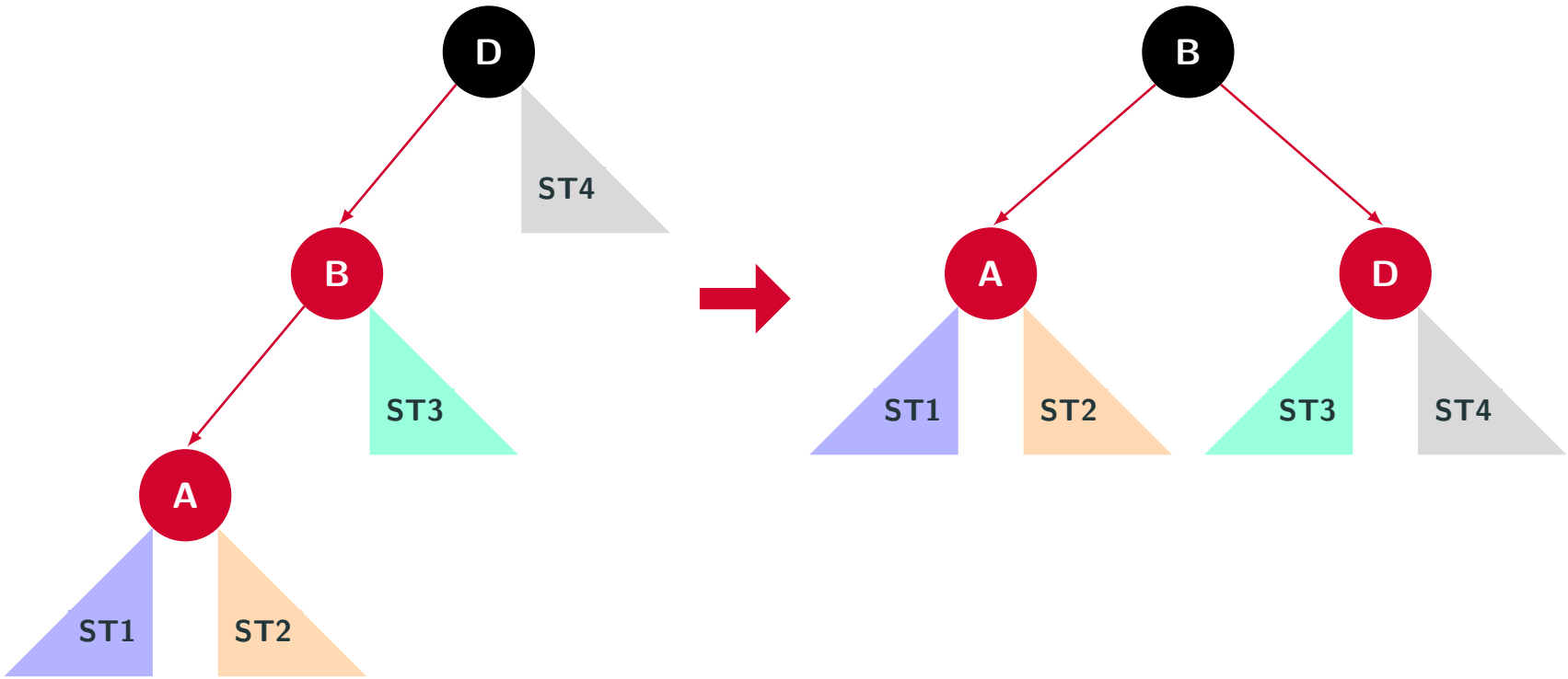


- Flip colours of the nodes, which is equivalent to sending the middle key of this 4-Node to the parent 2/3-Node in the 2-3 Search Tree.

- Continue recursively upwards with the parent 2/3-Node.

2nd case

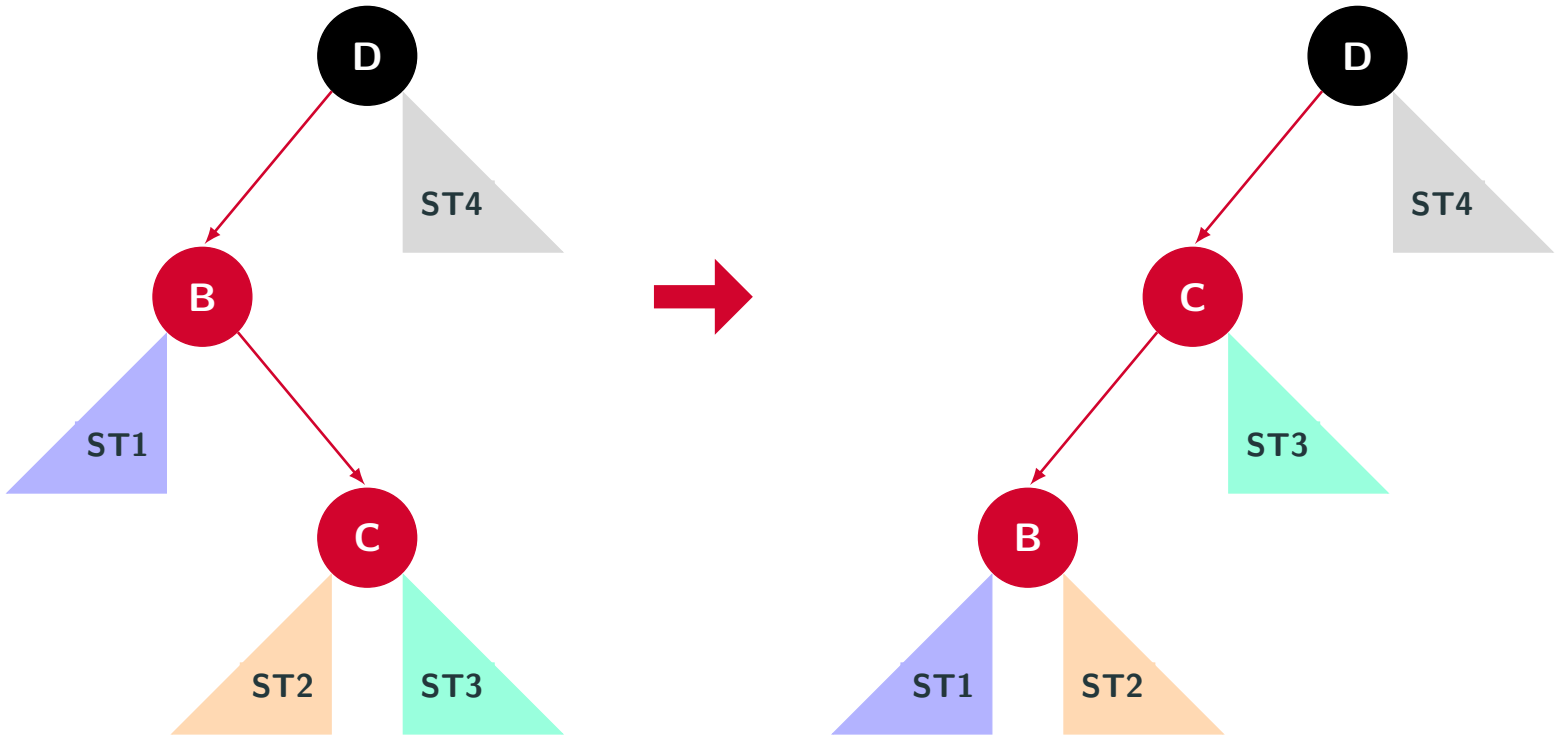
- If the new red key is smaller than both previous keys, for example:



- Perform a right rotation to have the middle key at the top, so converting it into the 1st case.

3rd case

- If the new red key is in between the previous keys, for example:



- Perform a left rotation on B to turn into 2nd case.

A few notes

- All rotations are performed along red edges, so they preserve the perfect black balance of the RBT.
- The colour flipping operation also preserve the perfect black balance of the RBT as the change of both children from red to black is balanced by the change of the parent from black to red.
- The 3rd case of 3-Nodes and the fix of 2-Nodes can be unified as they both trigger a left rotation.

The overall adjustment procedure

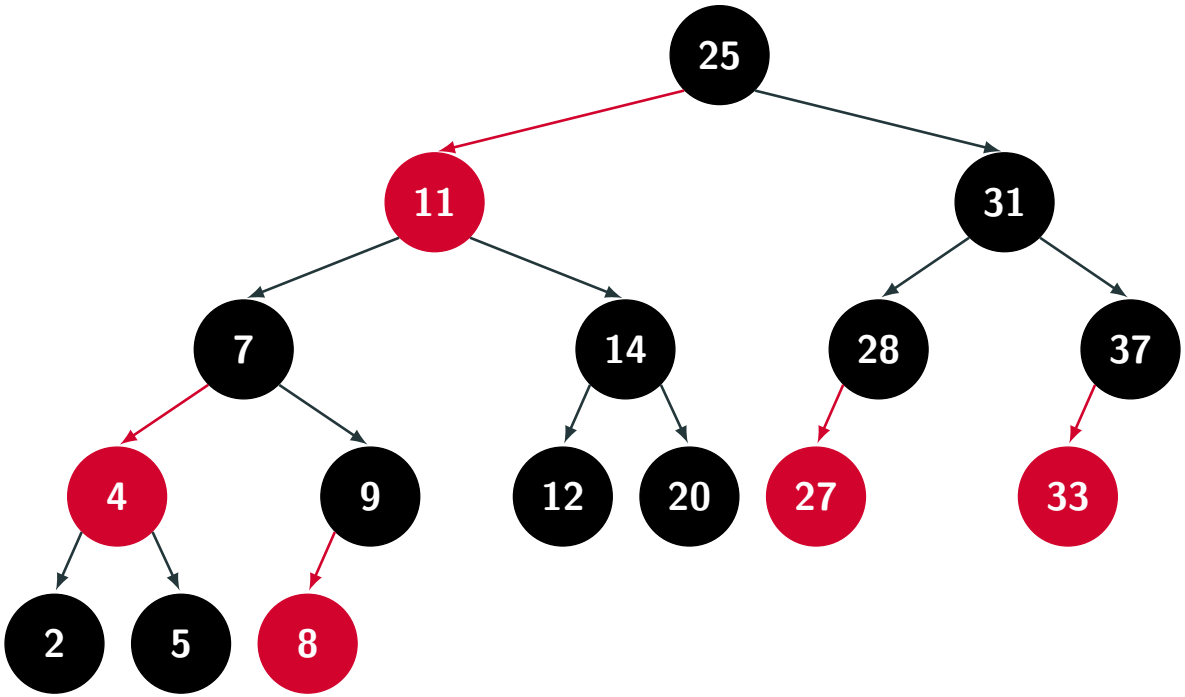
Moving from the point of insertion upwards, in the current node do:

1. If it only has one red child and that child is in the right, rotate left.
2. If the left child is red and its left child is also red, rotate right.
3. If both children are red, flip all colours of the three nodes (and their incoming edges).
4. (Optional Early Stop Condition): If the current node is black and none of the above conditions were true for the node, stop.
5. Proceed recursively to the parent node unless it is already the root node, in which case you just paint it black and finish.

Let's practice using the same sequence of insertions as in the 2-3 Search Tree example, but now using the RBT adjustment procedure.

Let's insert some keys . . .

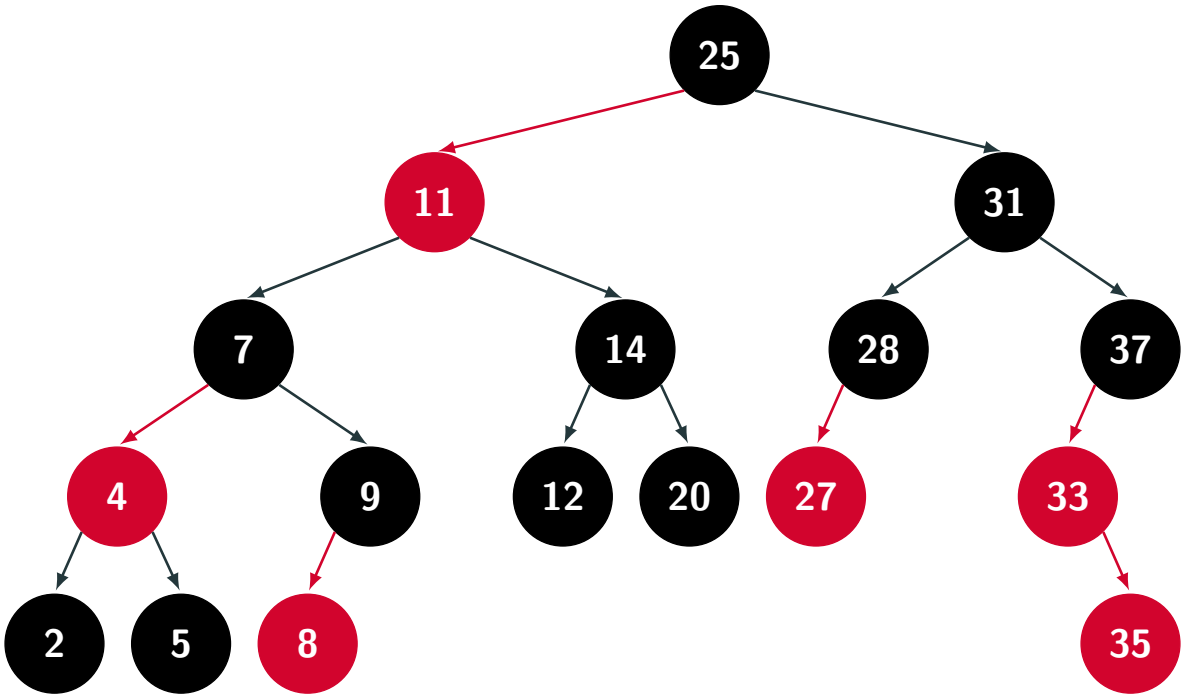
We start with 27:



The insertion point is node 28, and no adjustment condition is triggered.

Let's insert some keys . . .

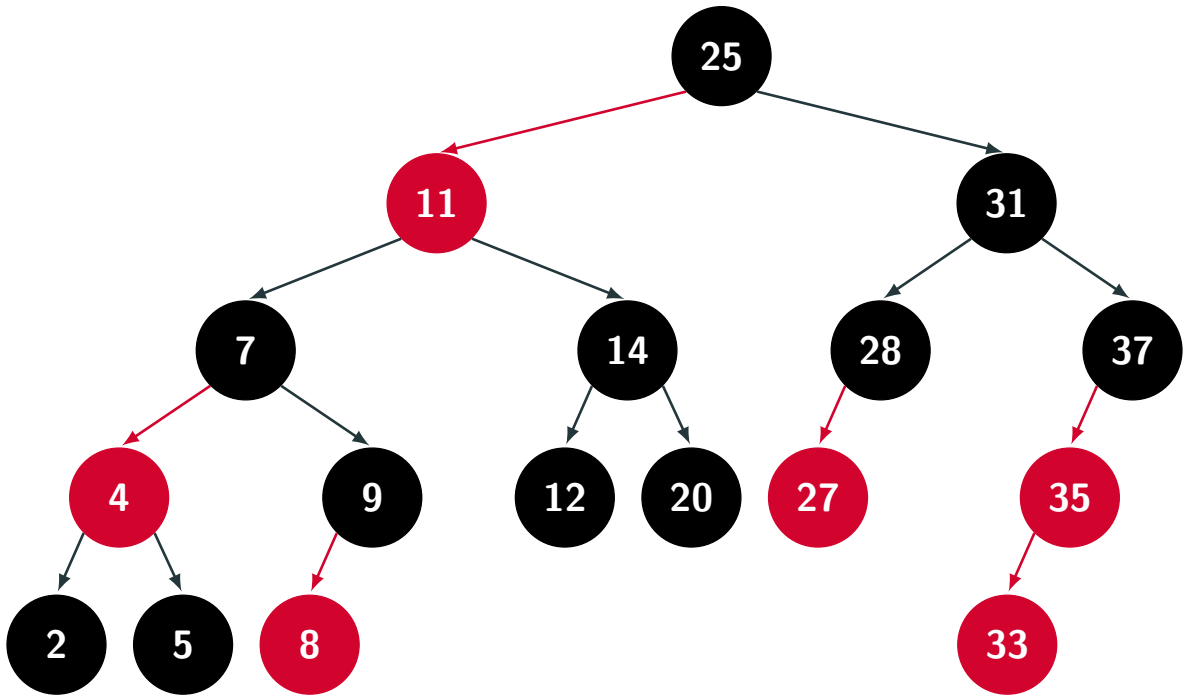
Then we insert key 35:



The point of insertion is node 33, and a left rotation (Condition 1) is triggered there.

Let's insert some keys . . .

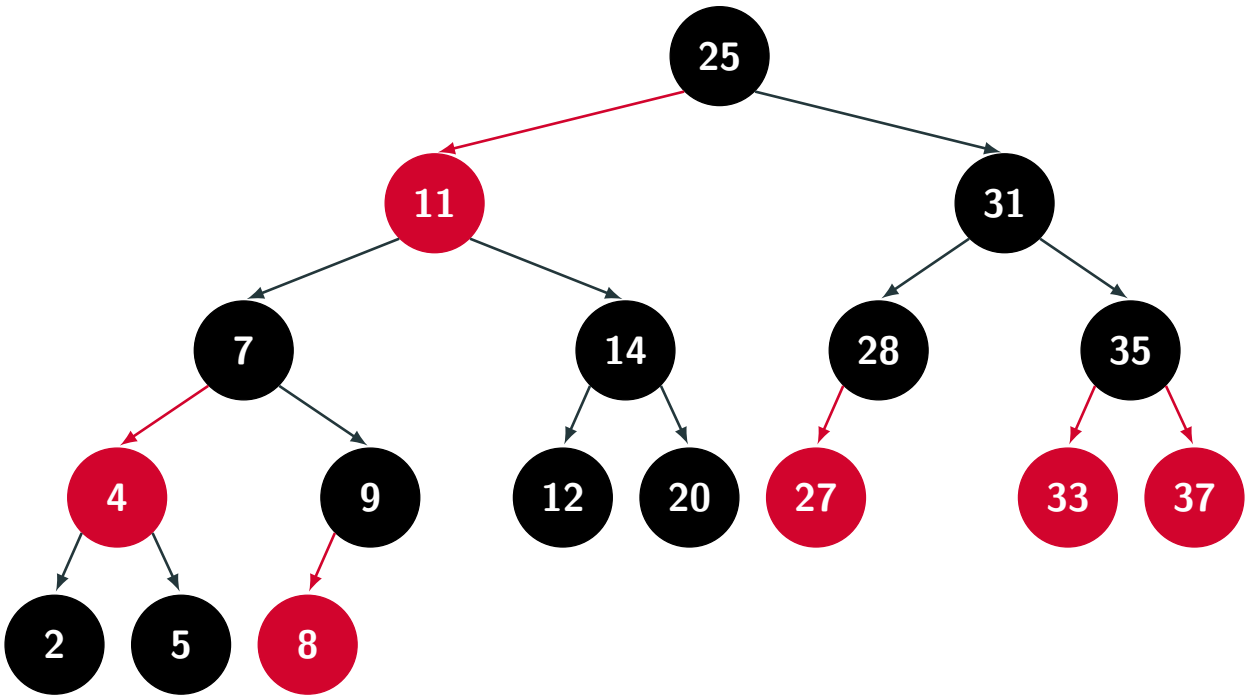
Doing the left rotation on node 33:



When we move to node 37 a right rotation (Condition 2) is triggered.

Let's insert some keys . . .

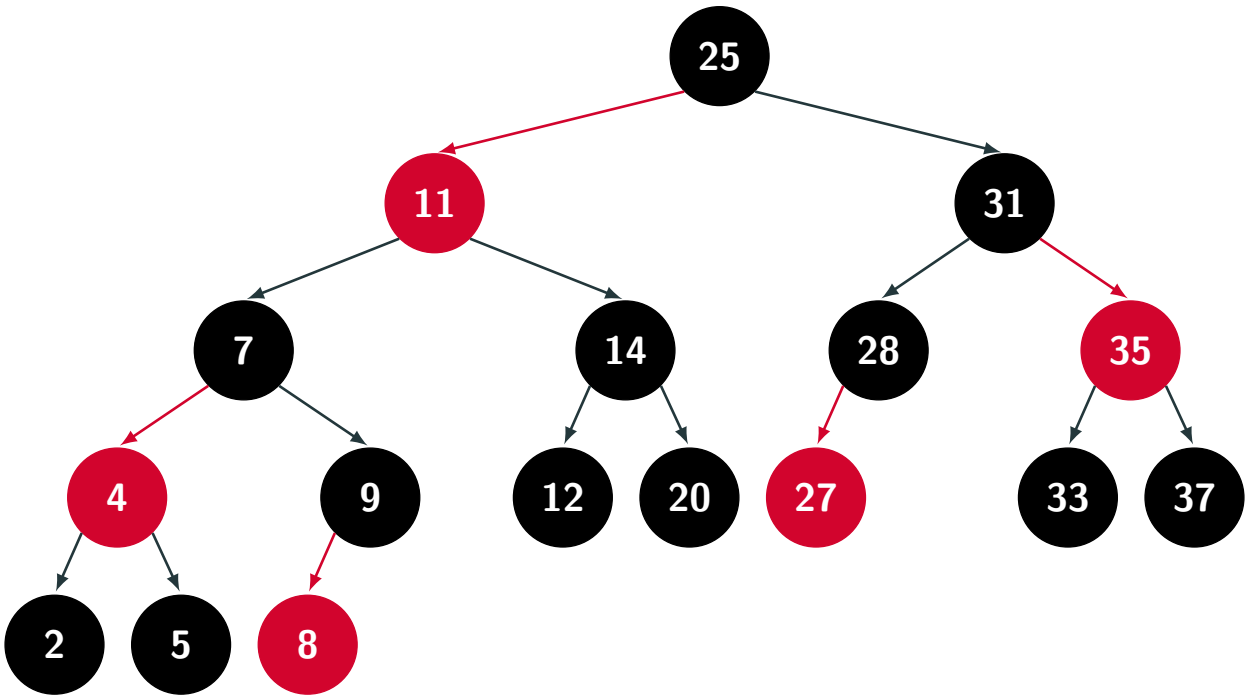
Doing the right rotation on node 37:



On node 35 a flipping of the colours of itself and its children (Condition 3) is triggered.

Let's insert some keys . . .

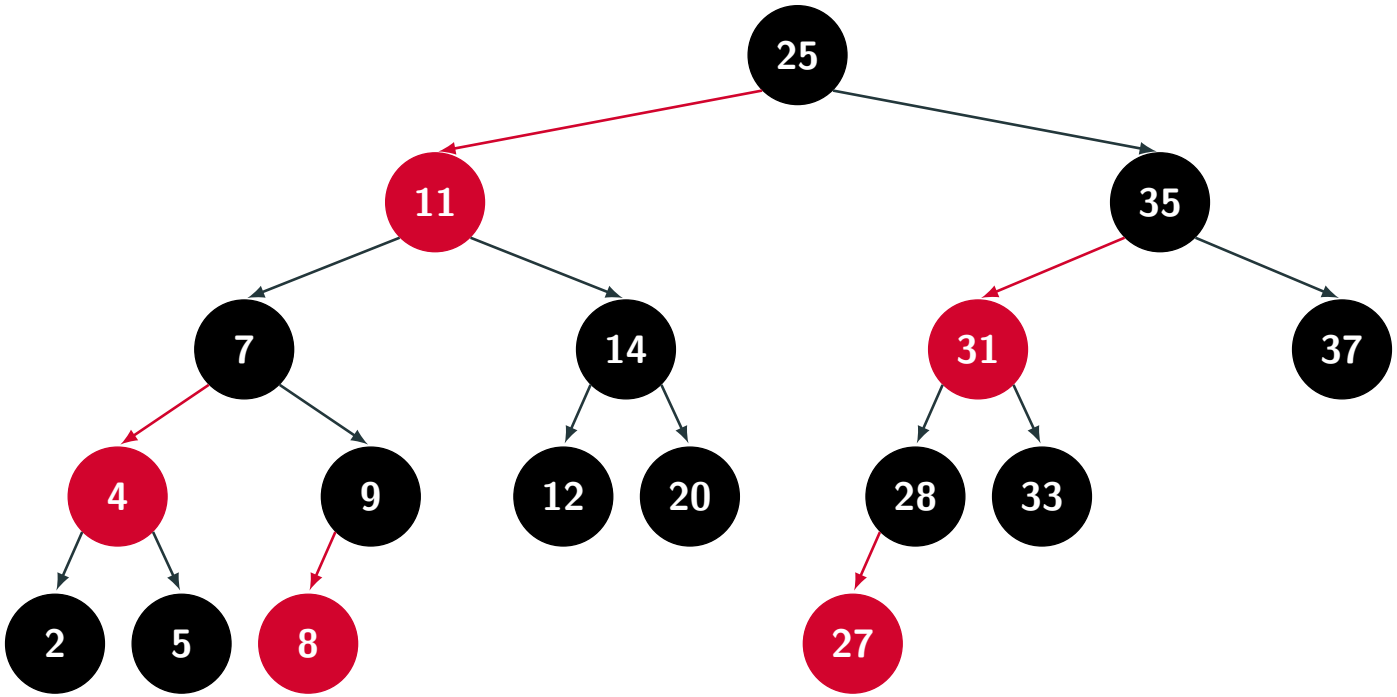
Flipping the colours:



On node 31 a left rotation (Condition 1) is executed.

Let's insert some keys . . .

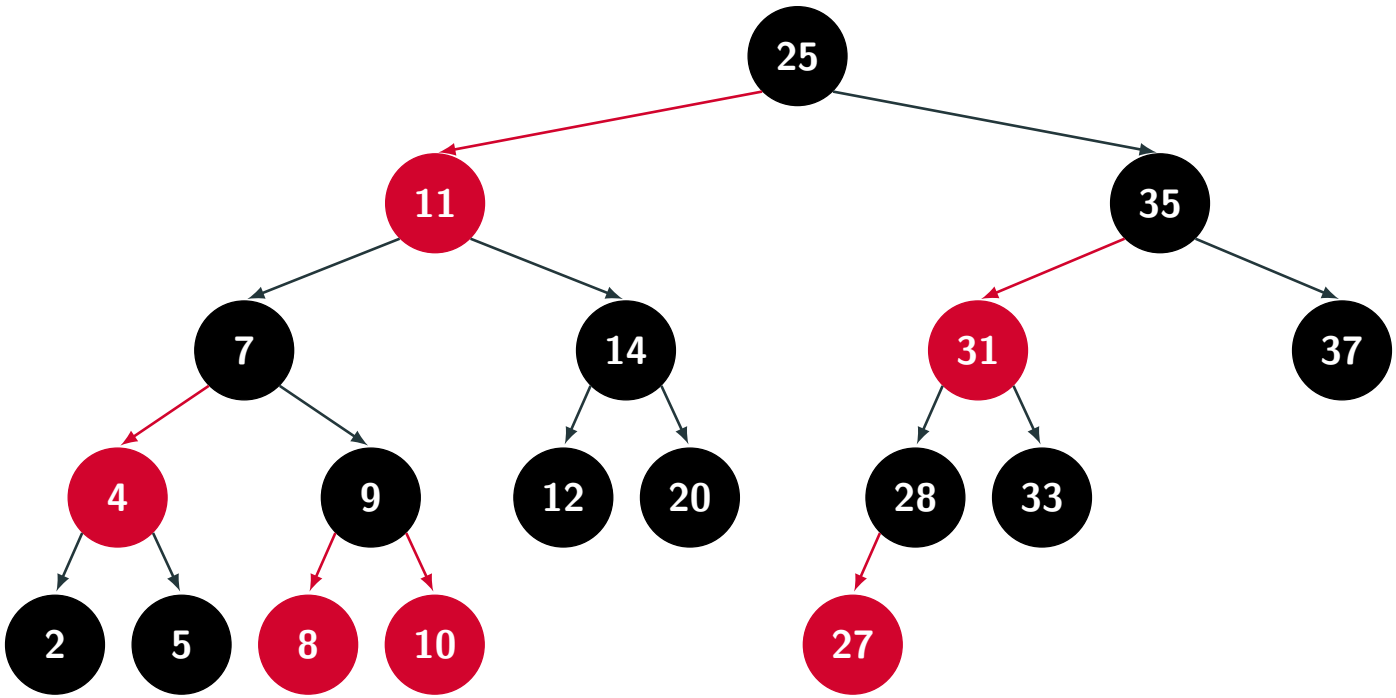
Doing the left rotation on node 31:



As the early stop condition holds on node 35 (Condition 4), we are done.

Let's insert some keys . . .

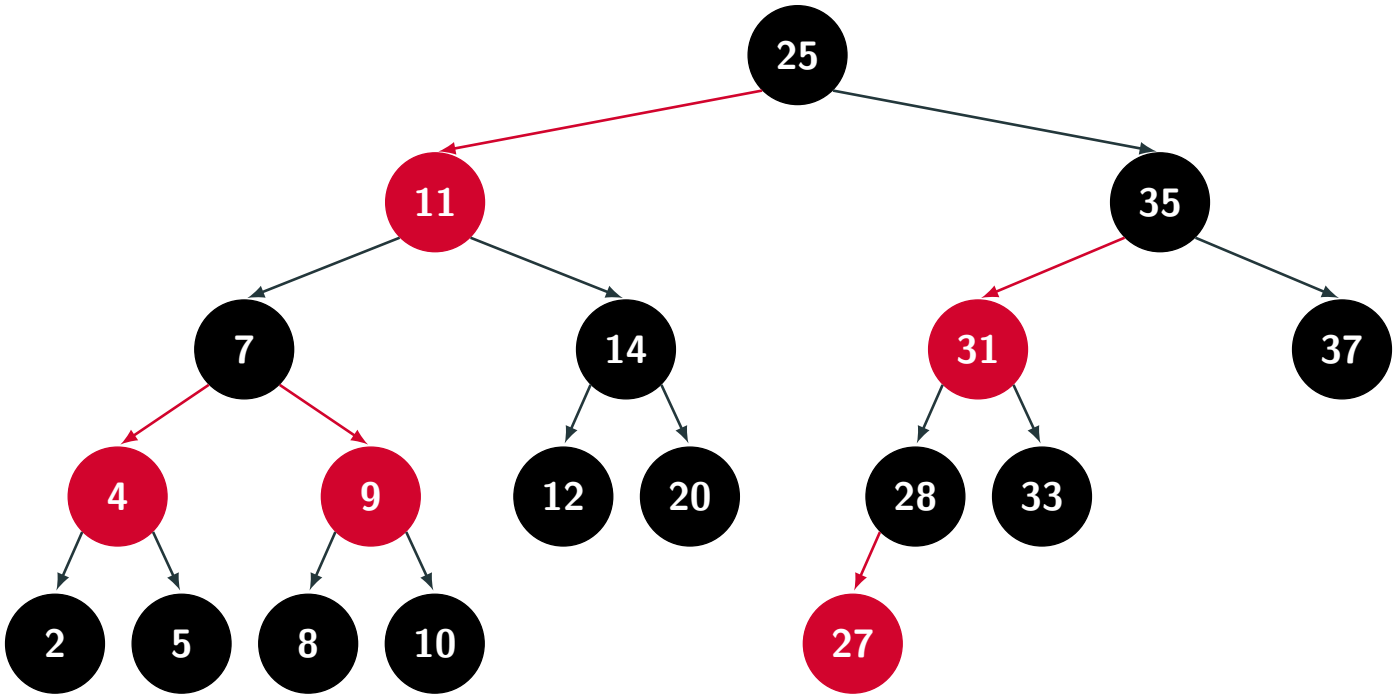
Finally, let's insert key 10:



The point of insertion is node 9, and it triggers a flipping of the colours of itself and its children (Condition 3).

Let's insert some keys . . .

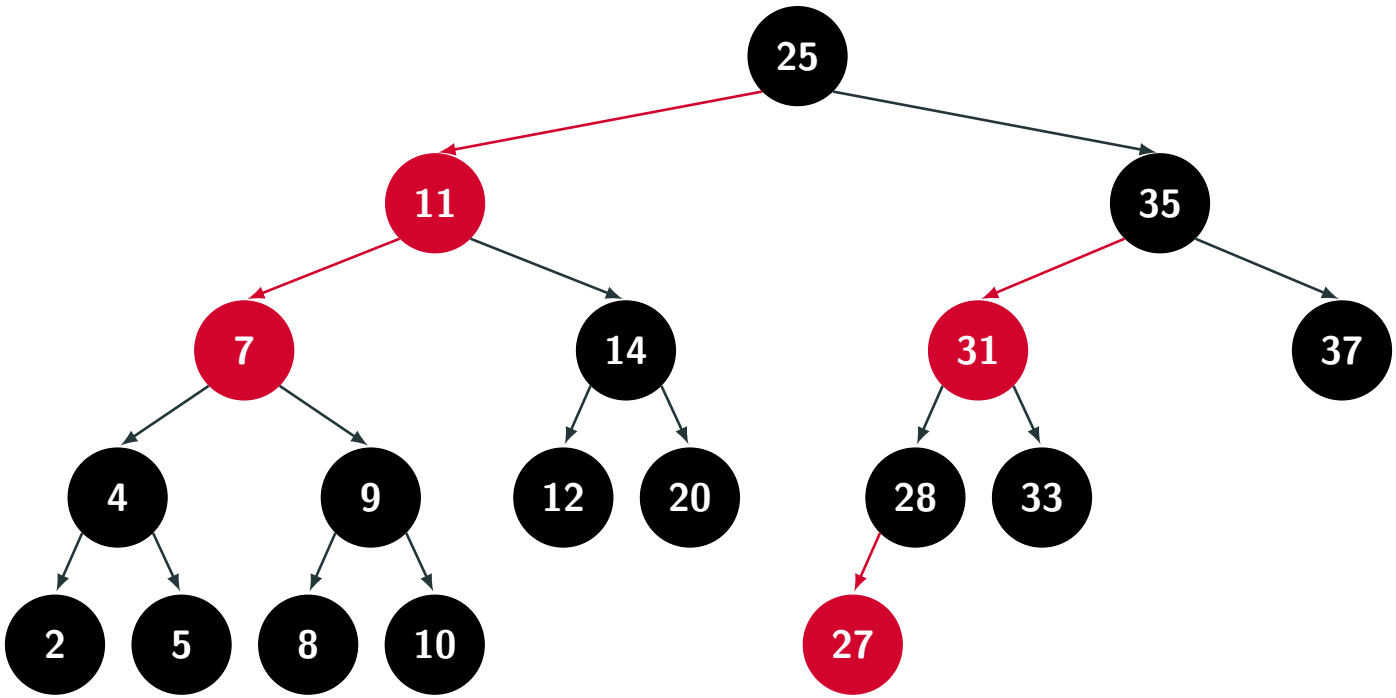
Flipping the colours:



On node 7, a flipping of the colours of itself and its children (Condition 3) is triggered again.

Let's insert some keys . . .

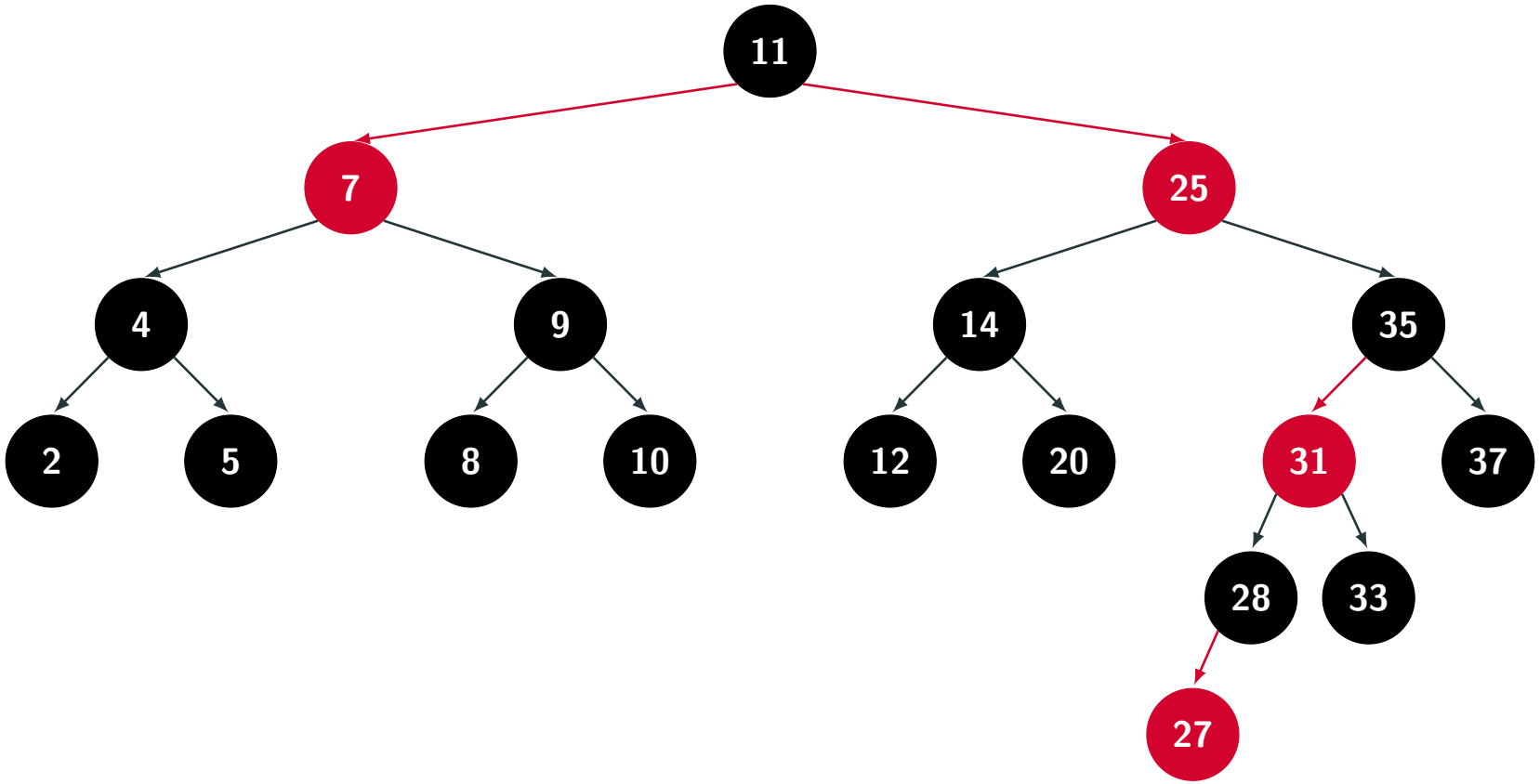
Flipping the colours:



No adjustment condition is true on node 11, but on node 25 a right rotation (Condition 2) is executed.

Let's insert some keys . . .

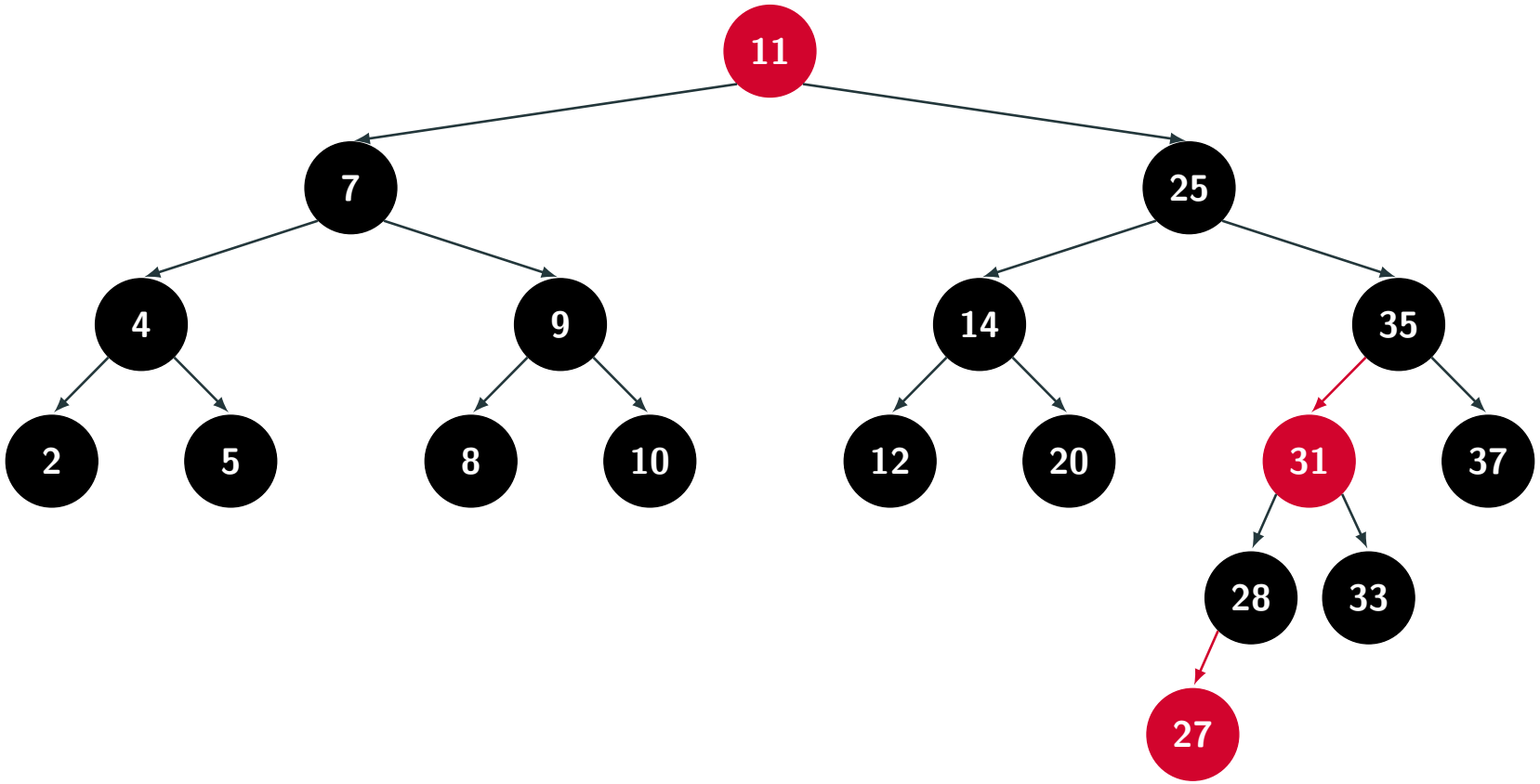
Performing the right rotation on node 25:



On node 11, a flipping of the colours of itself and its children (Condition 3) is triggered.

Let's insert some keys . . .

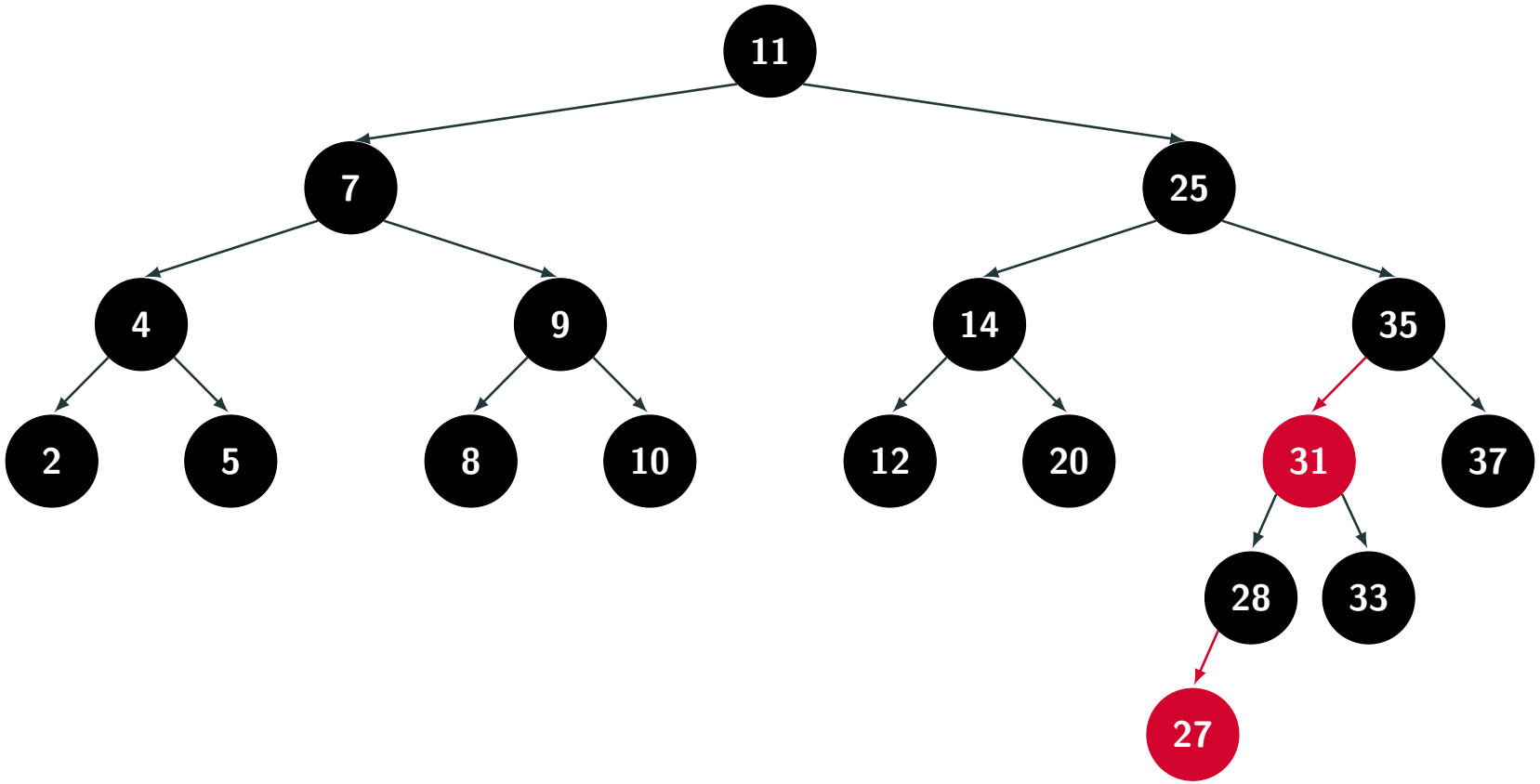
Flipping the colours:



Finally as node 11 is the root, its colour is switched back to black (Condition 5) to finalise the adjustments.

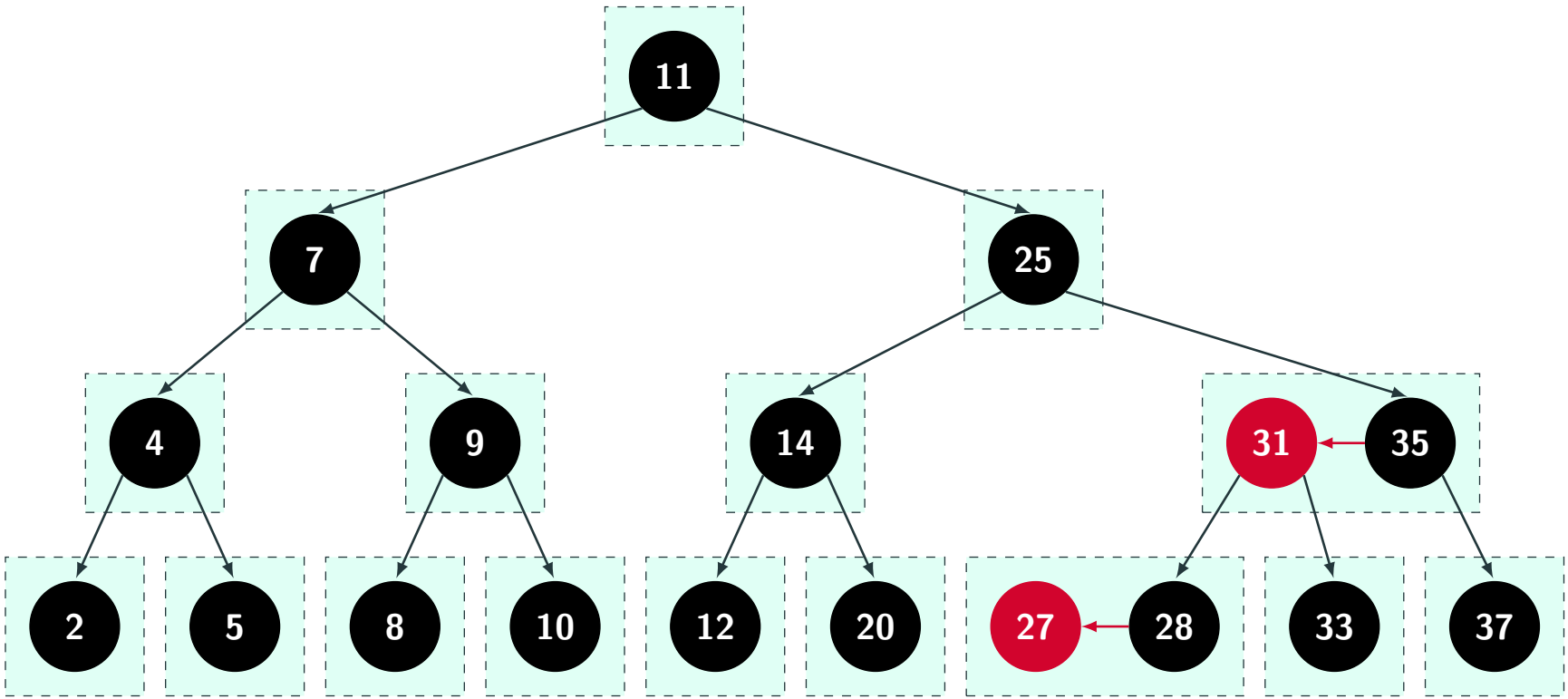
Let's insert some keys . . .

Flipping the root to black to finish:



Final result is equivalent

Flattening the tree, we can see that the same 2-3 Search Tree as before would be obtained:



Final notes about RBT

- The deletion procedure of RBT also explores the one-to-one correspondence with 2-3 Search Trees.

- The fact that search, insertion and deletion operations in a Left-Leaning Red-Black Trees have time complexity $O(\log(n))$ follows easily from its one-to-one correspondence with Perfectly Balanced 2-3 Search Trees.

AVL Trees versus RBT

- In terms of asymptotical worst-case time complexity, AVL and Red-Black Trees are equivalent.
- In concrete efficiency terms, when compared with other self-balancing BST data structures such as RBT, AVL Trees are more strictly balanced, which makes them a good choice for applications that are very intensive on lookups.
- RBT perform insertion and deletion operations more efficiently in concrete terms as they are less strictly balanced and therefore require less rotations while doing those operations.

Reading

- Course Notes: Chapter 10
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - Robert Sedgewick and Kevin Wayne, *Algorithms*: Chapter 3
  - CLRS: Chapters 12 and 13
  - Rou: Chapter 11

Concluding remarks

- Take home message: AVL Trees, 2-3 Search Trees and RBTs offer $O(\log(n))$ time complexity for search, insertion and deletion operations.
- Things to do:
  - Practice balancing those search trees using pen and paper.
  - Implement those search trees.
- Coming up next:
  - Overview and revision