Data Structures

Search Trees

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

The problem

Search

We would like to maintain items with keys and except from add/remove to also be able to quickly search an item based on a key.

Example

Assume items are bank transactions we would like to search based on the date.

Search

We can implement a data structure for searching in multiple ways.

Our goal is to implement the search operations efficiently while keeping the remaining operations also efficient.

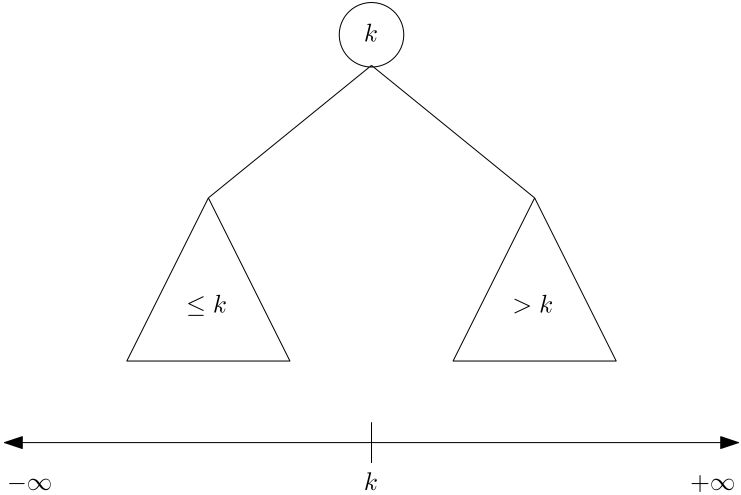
e.g. searching on a linked list is not efficient!

Binary Search Tree (BST)

Definition

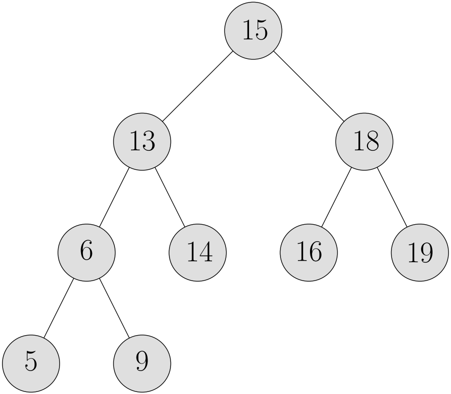
A **binary search tree** is a binary tree where each node is associated with a key with the additional property that the key of a node is larger (or equal) from the keys of all nodes in its left subtree, and smaller that all the keys of the nodes in its right subtree.

Binary Search Tree (BST)



Example

Binary Search Tree (BST)



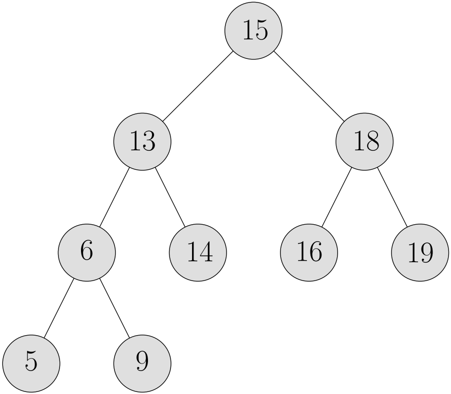
Searching on a BST

Algorithm

- start from the root
- if we are at a node which has the same key as the one we are looking for, then return the node
- otherwise go left or right depending on the comparison between the key we are looking for and the key of the node

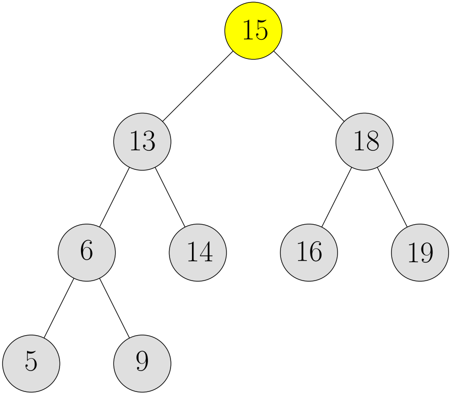
Searching on a BST

Searching for 9



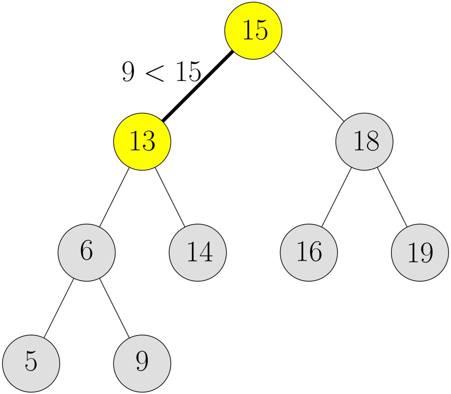
Searching on a BST

Searching for 9



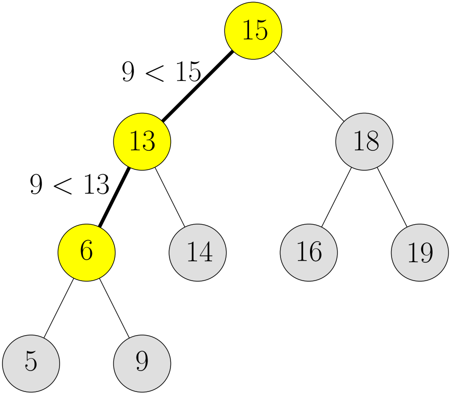
Searching on a BST

Searching for 9



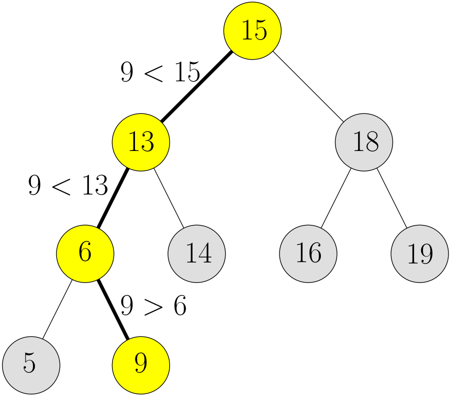
Searching on a BST

Searching for 9



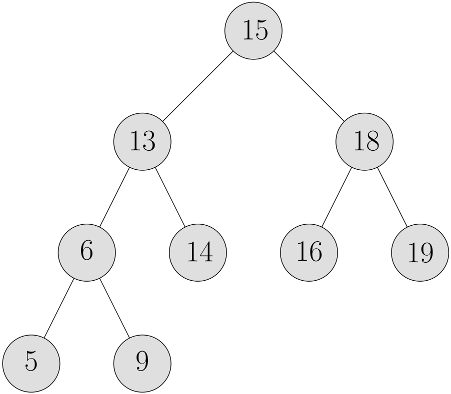
Searching on a BST

Searching for 9



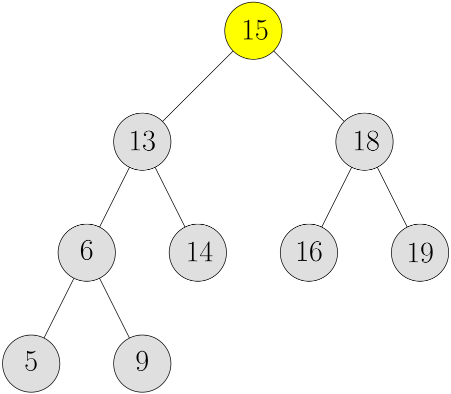
Searching on a BST

Searching for 17



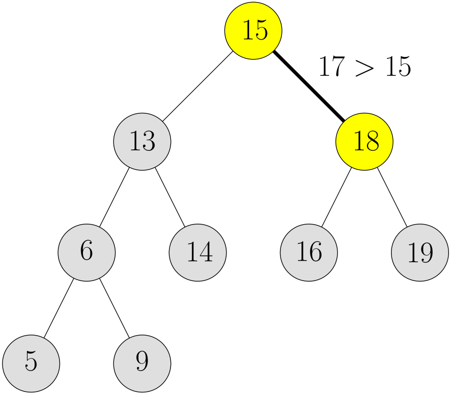
Searching on a BST

Searching for 17



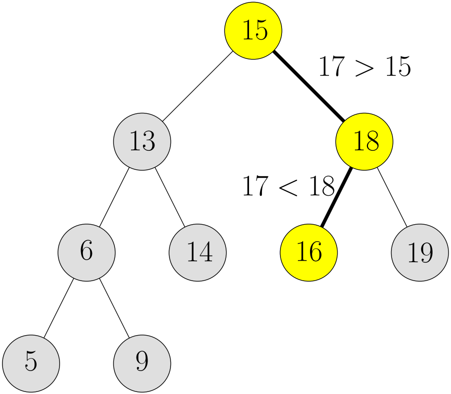
Searching on a BST

Searching for 17



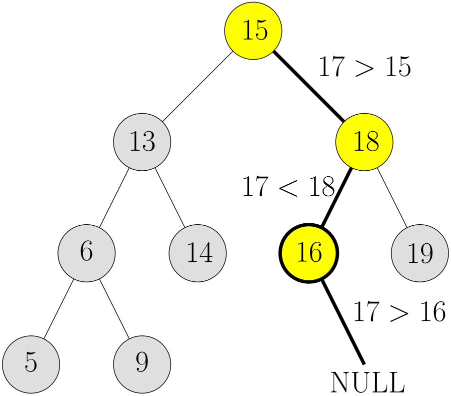
Searching on a BST

Searching for 17



Searching on a BST

Searching for 17



Inorder Traversal

Rule

First visit the left subtree recursively, then the root then the right subtree recursively.

Returns nodes in non-decreasing key order.

The traversal needs linear time $\mathcal{O}(n)$ where n are the number of items in the tree.

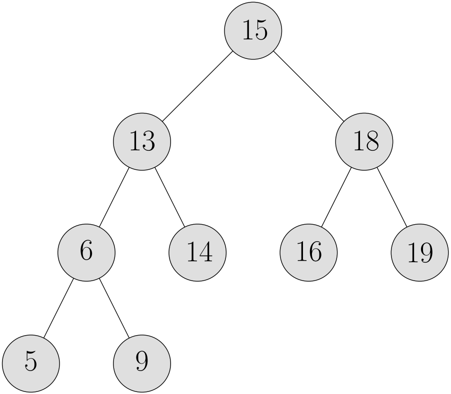
Insertion on a BST

Algorithm

- first perform a search
- if the key is found, do nothing
- otherwise add a new node with the new key at the point where the search ended

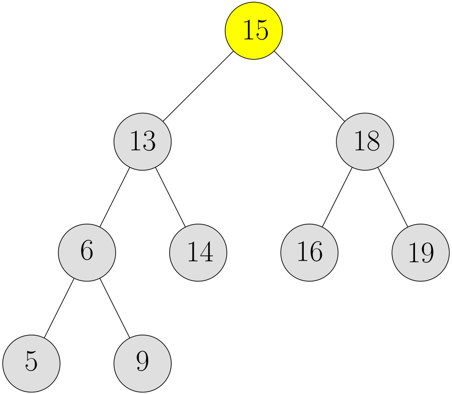
Insertion on a BST

Insertion of 17



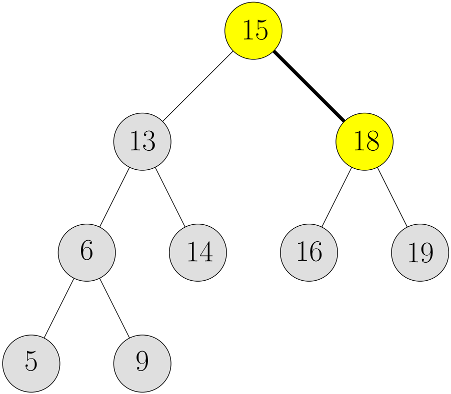
Insertion on a BST

Insertion of 17



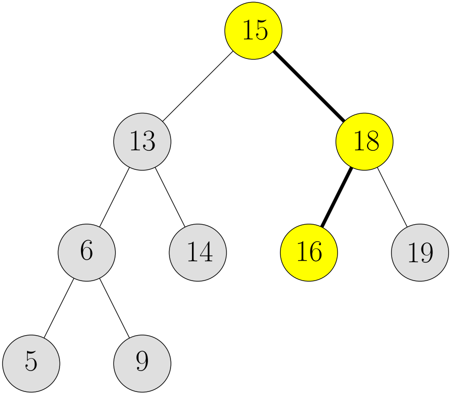
Insertion on a BST

Insertion of 17



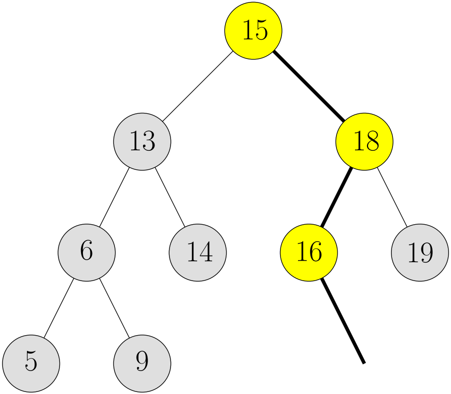
Insertion on a BST

Insertion of 17



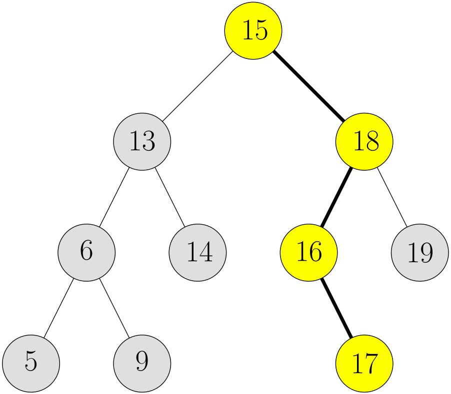
Insertion on a BST

Insertion of 17



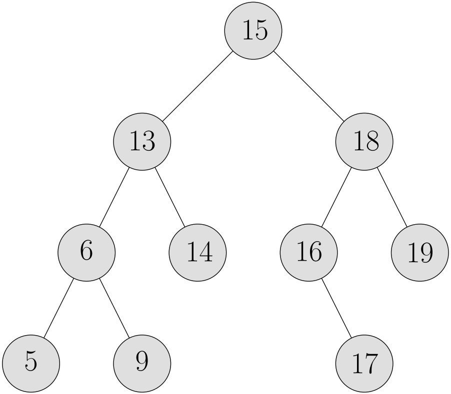
Insertion on a BST

Insertion of 17



Insertion on a BST

Insertion of 17



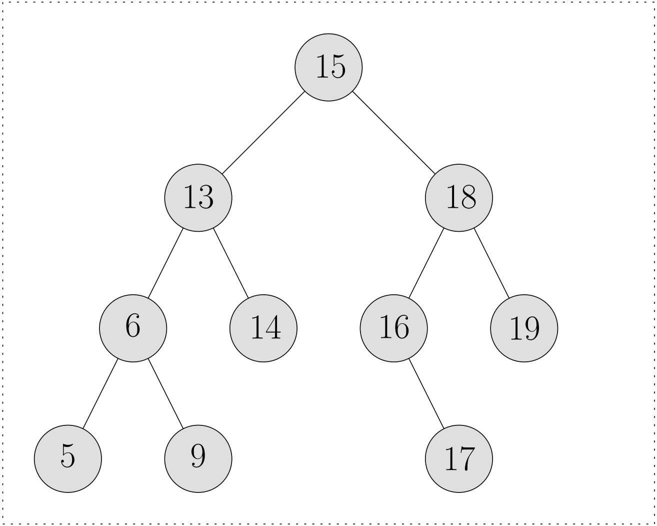
Deletion from a BST

Algorithm

- first perform a search
- if the key is not found, do nothing
- otherwise there are 3 cases based on the number of children of the node we want to delete

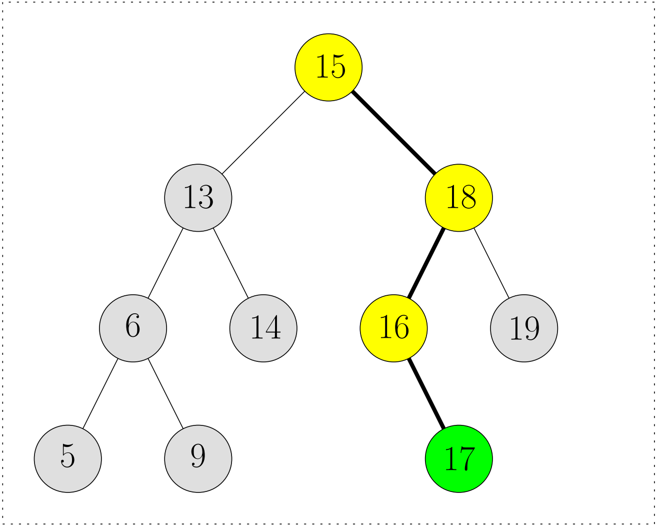
Deletion from a BST

Case 1: Deletion of 17 (no child)



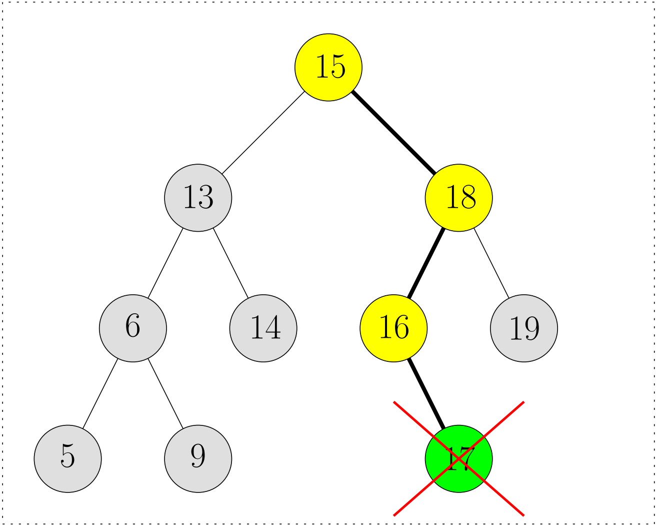
Deletion from a BST

Case 1: Deletion of 17 (no child)



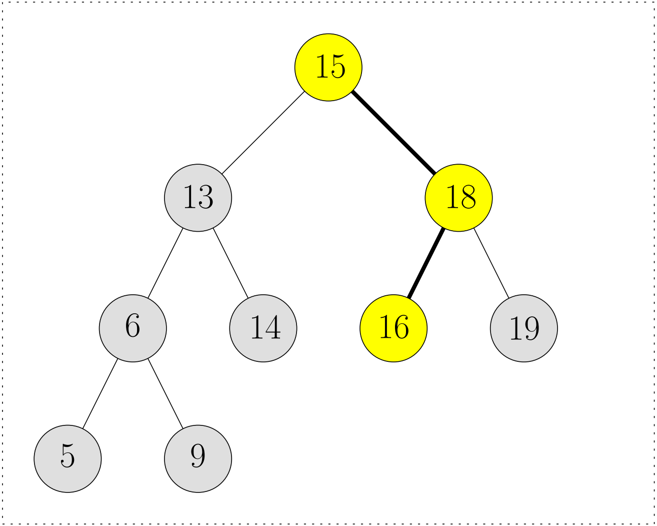
Deletion from a BST

Case 1: Deletion of 17 (no child)



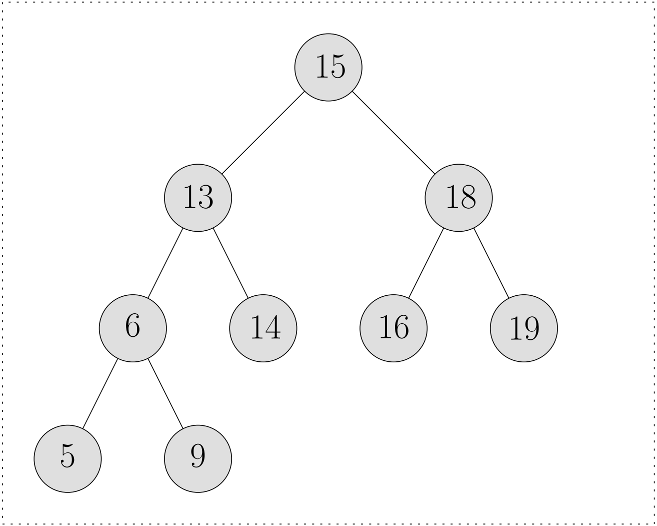
Deletion from a BST

Case 1: Deletion of 17 (no child)



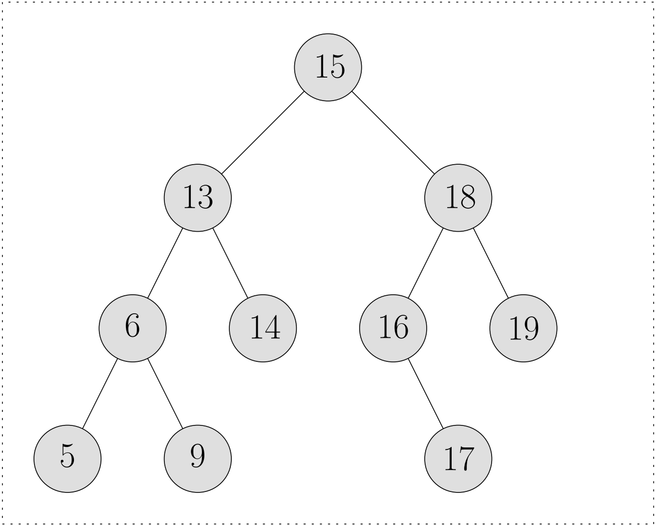
Deletion from a BST

Case 1: Deletion of 17 (no child)



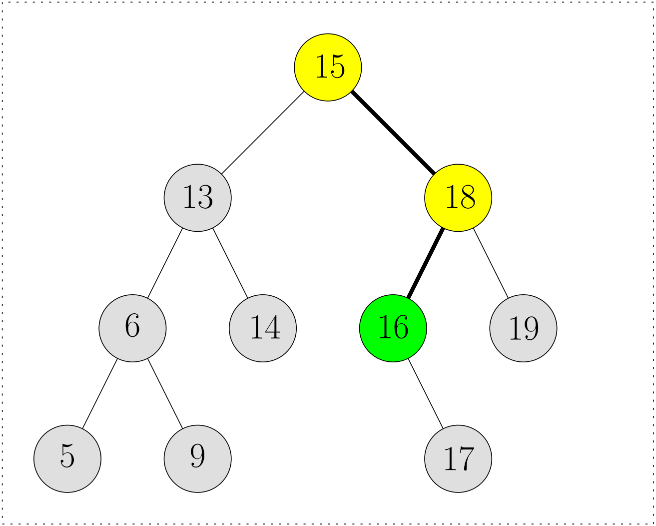
Deletion from a BST

Case 2: Deletion of 16 (one child)



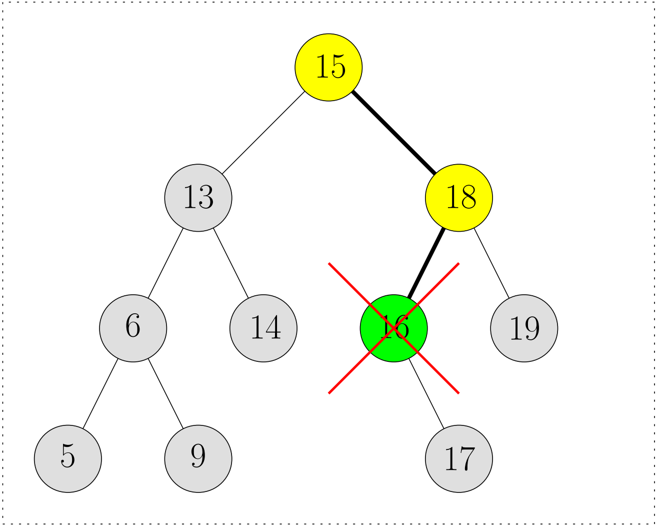
Deletion from a BST

Case 2: Deletion of 16 (one child)



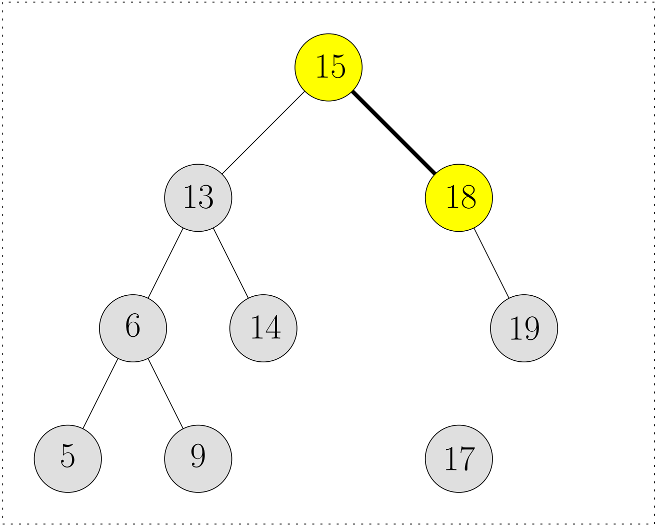
Deletion from a BST

Case 2: Deletion of 16 (one child)



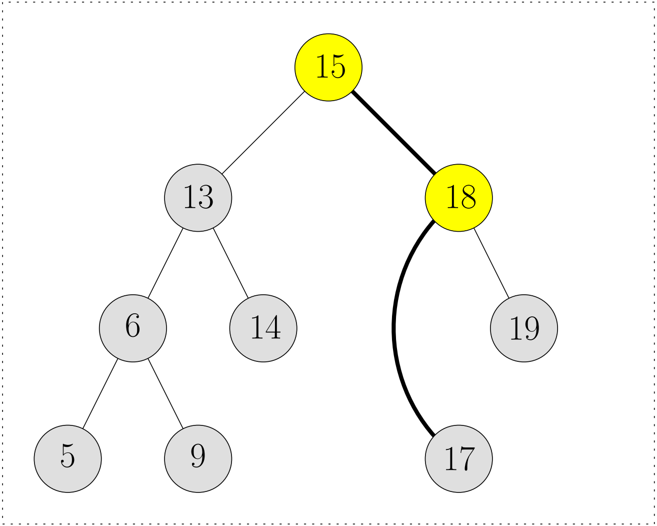
Deletion from a BST

Case 2: Deletion of 16 (one child)



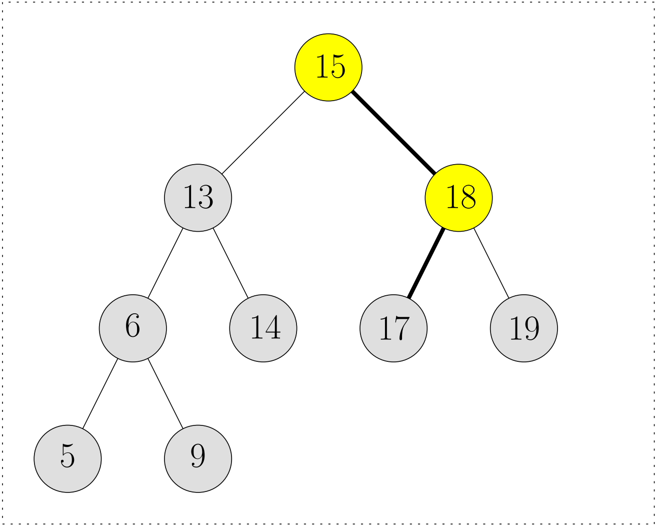
Deletion from a BST

Case 2: Deletion of 16 (one child)



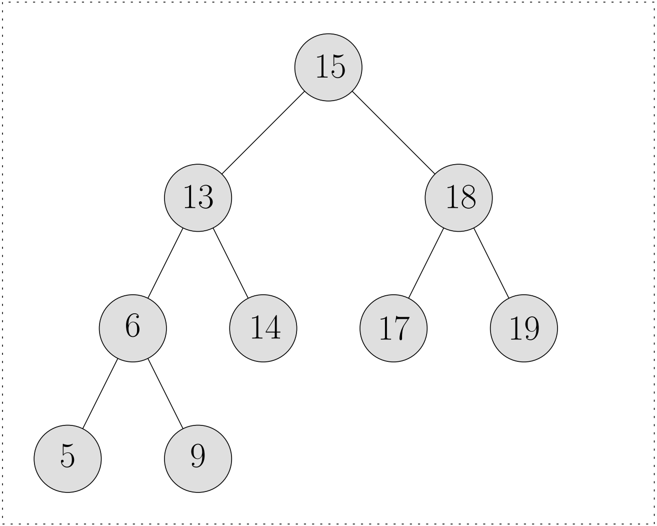
Deletion from a BST

Case 2: Deletion of 16 (one child)



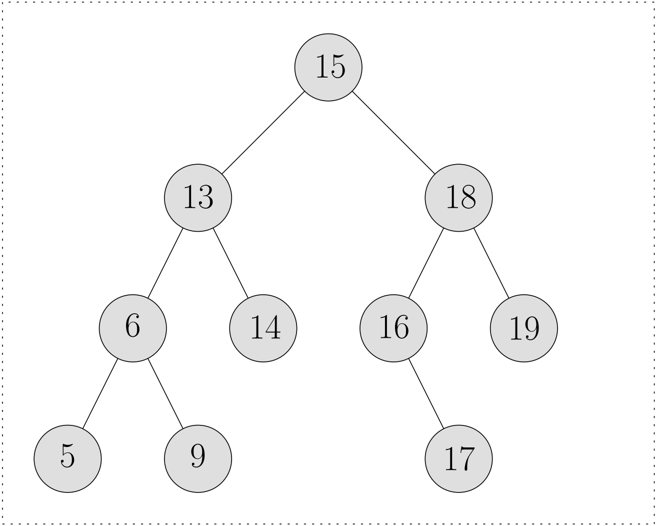
Deletion from a BST

Case 2: Deletion of 16 (one child)



Deletion from a BST

Case 3: Deletion of 15 (two children)



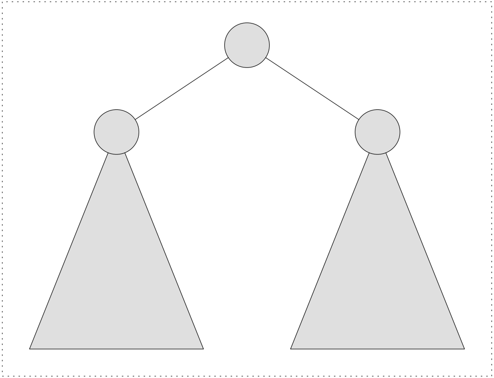
Calculate the successor

Useful routine for deletion

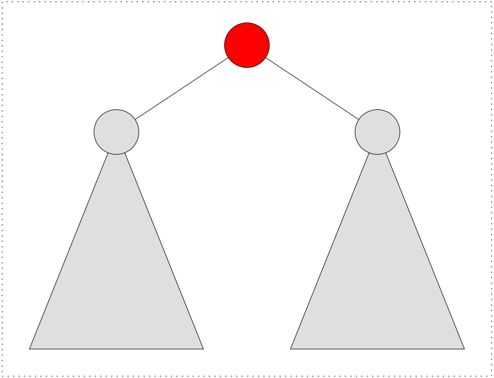
The problem

Given a subtree we would like to find the node with the successor key from the root of the subtree

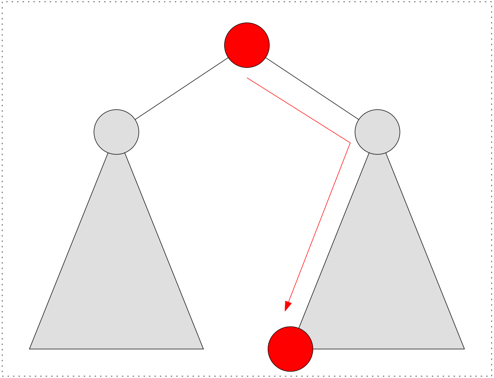
Finding the successor of the root key



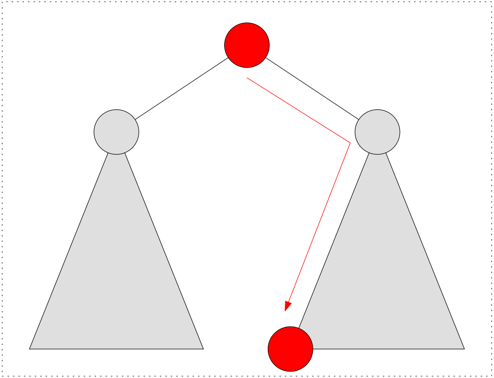
Finding the successor of the root key



Finding the successor of the root key



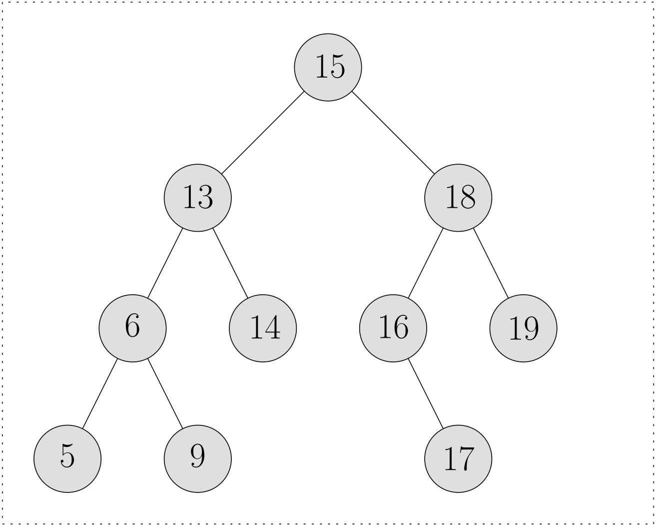
Finding the successor of the root key



The successor of the root has at most one child. (why?)

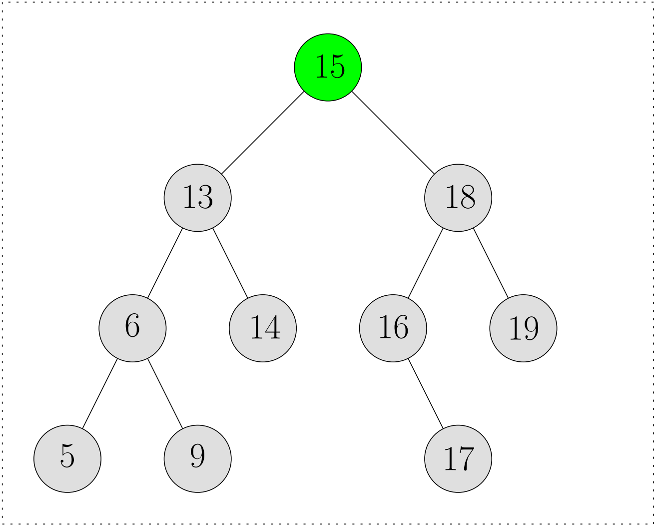
Deletion from a BST

Case 3: Deletion of 15 (two children)



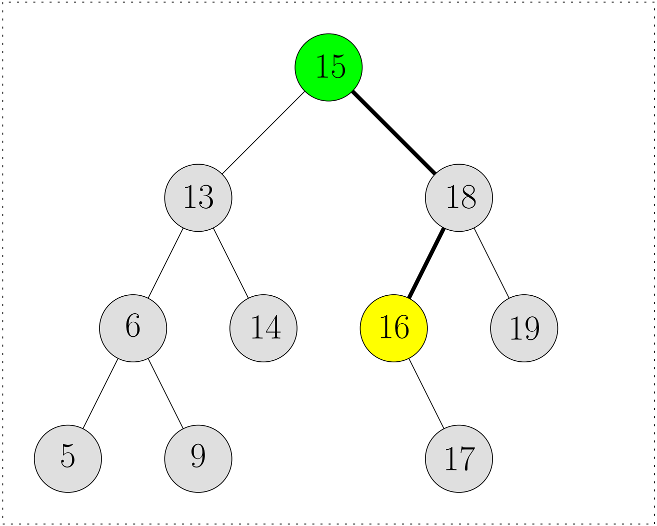
Deletion from a BST

Case 3: Deletion of 15 (two children)



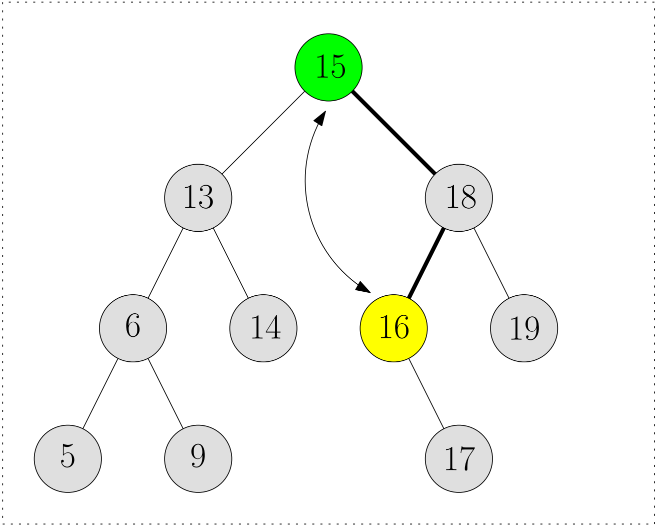
Deletion from a BST

Case 3: Deletion of 15 (two children)



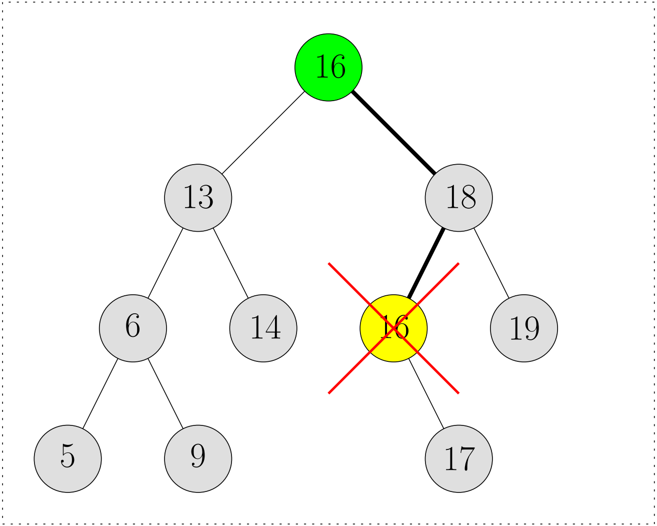
Deletion from a BST

Case 3: Deletion of 15 (two children)



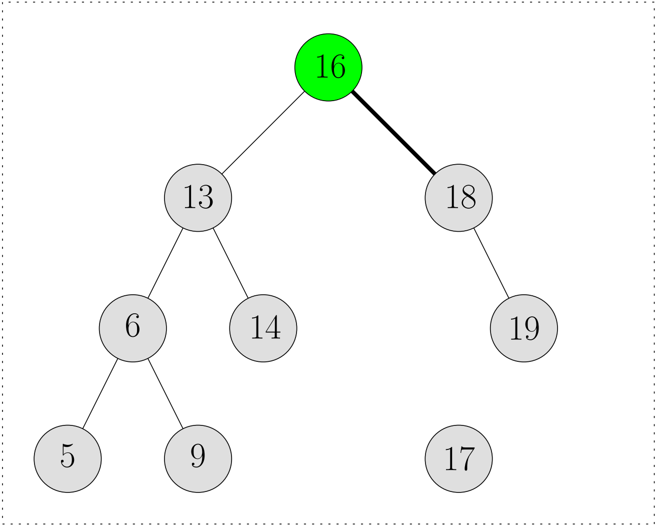
Deletion from a BST

Case 3: Deletion of 15 (two children)



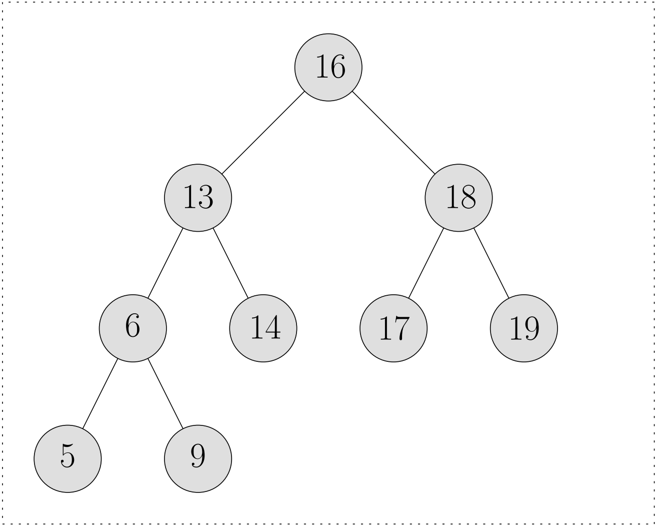
Deletion from a BST

Case 3: Deletion of 15 (two children)



Deletion from a BST

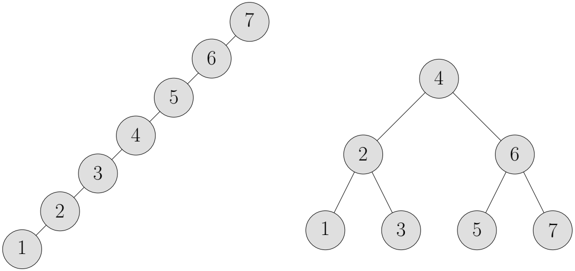
Case 3: Deletion of 15 (two children)



Is it fast?

All operations require time proportional to the height of the tree.

Problems



There are many trees for the same set of items

- worst case on the left: height $h = \mathcal{O}(n)$
- best case on the right: height $h = \mathcal{O}(\log n)$

The tree depends on the order that the insertions and deletions are performed, something that we cannot know beforehand

Balanced BSTs

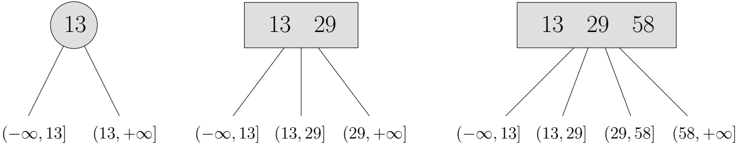
Search trees which are balanced at any point in time independently of the order that the insertions and/or deletions of items are performed.

$$h = \mathcal{O}(\log n)$$

2-3-4 Search Trees

top-down

We allow nodes which have 2, 3 or 4 children.



2-3-4 Search Trees

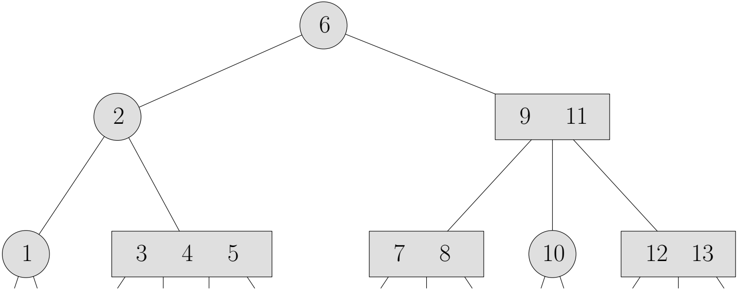
top-down

Definition

A **balanced 2-3-4 search tree** is a tree which:

- is either empty
- or is comprised of tree different node types: **2-nodes**, **3-nodes** **4-nodes**
- all the external nodes (nulls - null links) are at the exact same distance from the root

Example of a Balanced 2-3-4 Search Tree



Height of a Balanced 2-3-4 Search Tree

Theorem

A 2-3-4 search tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let h be the height of the tree.

Height of a Balanced 2-3-4 Search Tree

Theorem

A 2-3-4 search tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let h be the height of the tree.

Every level i has at least 2^i nodes.

Height of a Balanced 2-3-4 Search Tree

Theorem

A 2-3-4 search tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let h be the height of the tree.

Every level i has at least 2^i nodes.

Thus $n \geq 1 + 2 + 4 + \ldots 2^{h-1} = 2^h - 1$.

Height of a Balanced 2-3-4 Search Tree

Theorem

A 2-3-4 search tree with n keys has height $\mathcal{O}(\log n)$*.*

Proof.

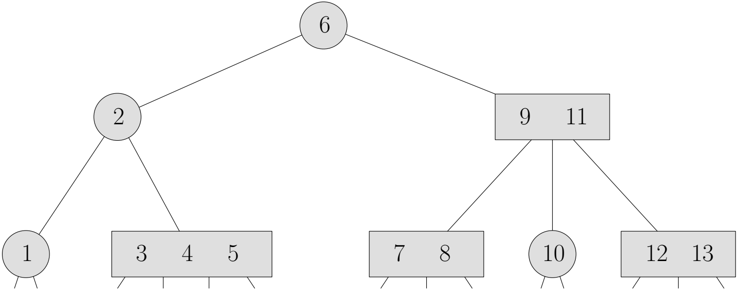
Let h be the height of the tree.

Every level i has at least 2^i nodes.

Thus $n \geq 1 + 2 + 4 + \ldots 2^{h-1} = 2^h - 1$.

Taking logarithms we get that $\log_2(n+1) \geq h$. □

Search on a Balanced 2-3-4 Search Tree



Search on a Balanced 2-3-4 Search Tree: Generalization of the BST algorithm

Insertion on a Balanced 2-3-4 Search Tree

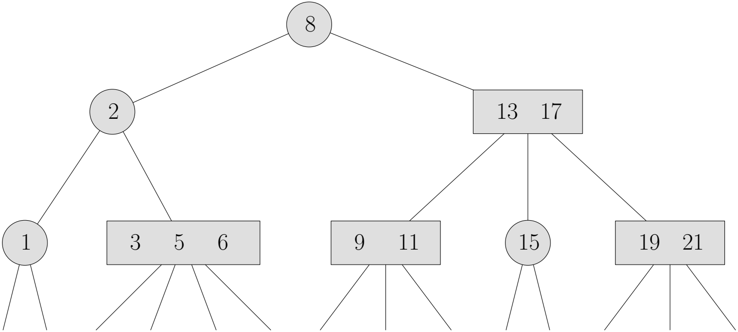
While maintaining the balance!

We search for the new key in order to find where we are supposed to add it.

- if the search ends on a 2-node we simply make it a 3-node
- if the search ends on a 3-node we make it a 4-node
- if the search ends on a 4-node
  - break the 4-node into two 2-nodes by sending the middle key upwards, and then
  - add the new key in one of the two 2-nodes (we might need to do the same procedure at the parent node, possibly multiple times all the way up to the root)

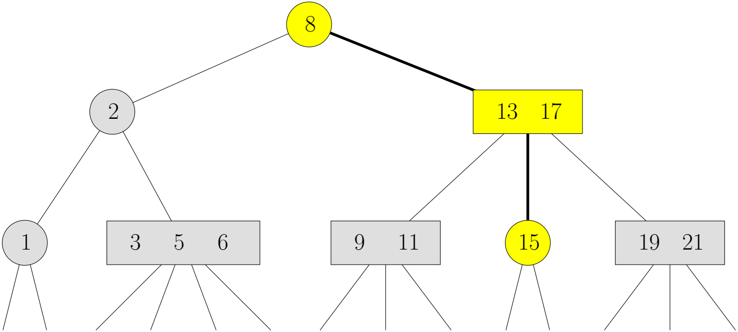
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 2-node (insertion of 14)



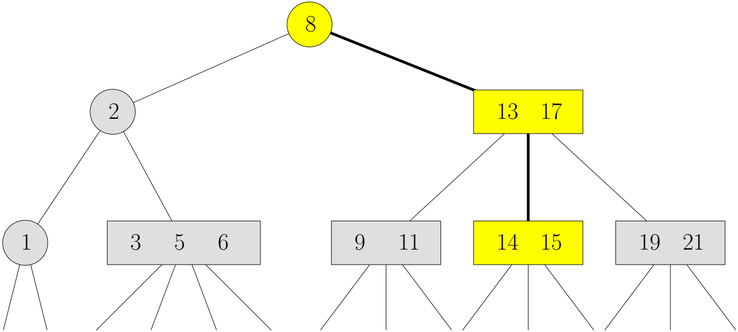
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 2-node (insertion of 14)



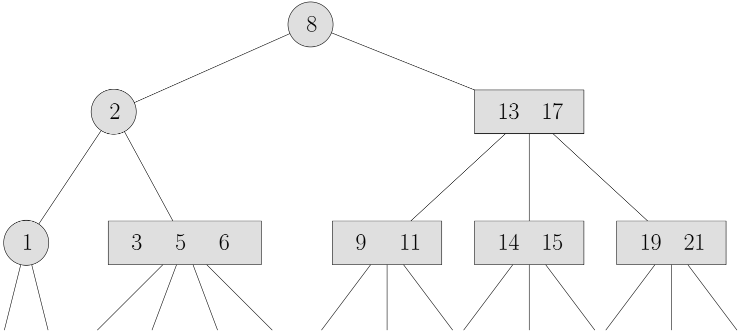
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 2-node (insertion of 14)



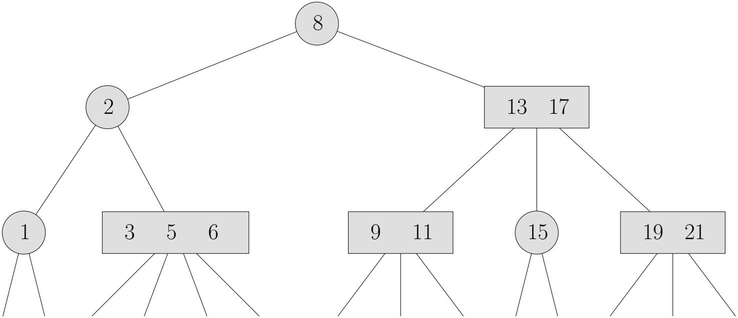
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 2-node (insertion of 14)



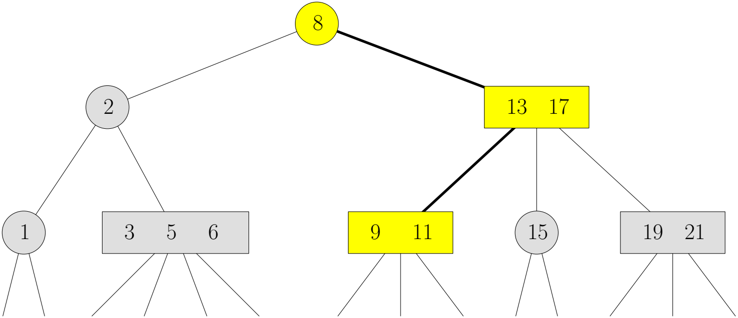
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 3-node (insertion of 10)



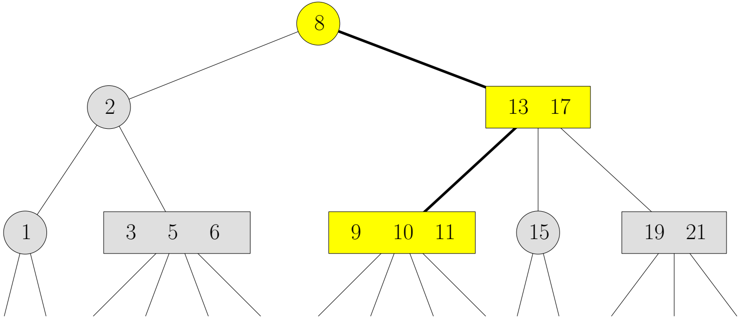
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 3-node (insertion of 10)



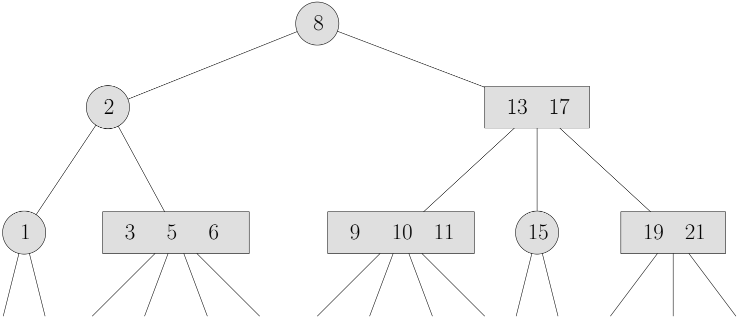
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 3-node (insertion of 10)



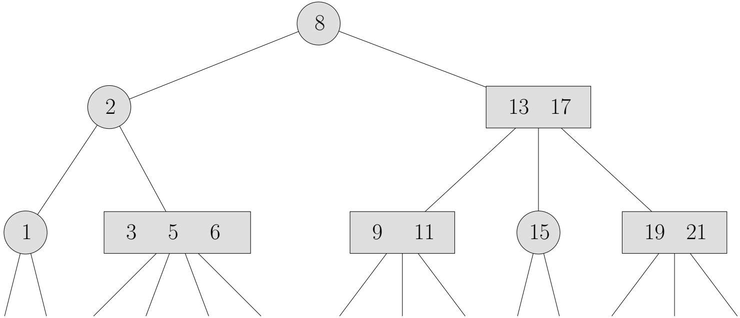
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 3-node (insertion of 10)



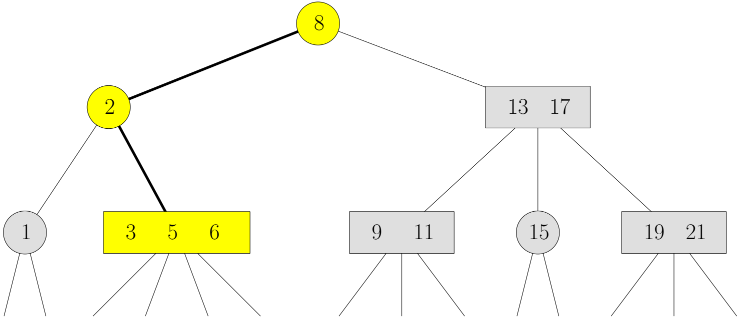
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



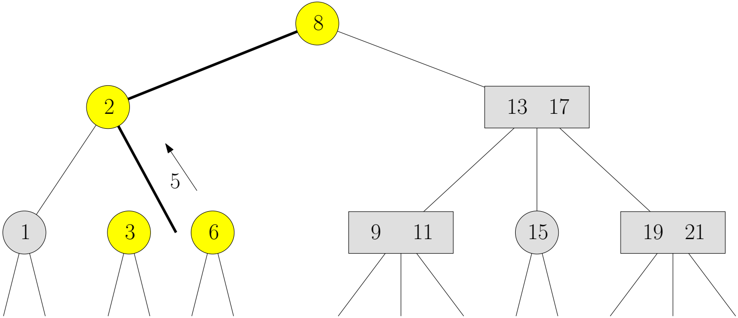
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



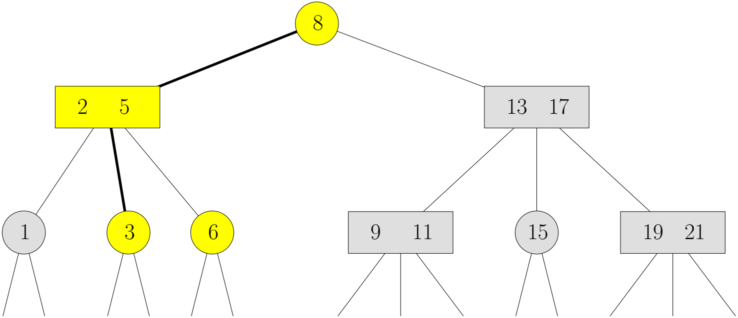
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



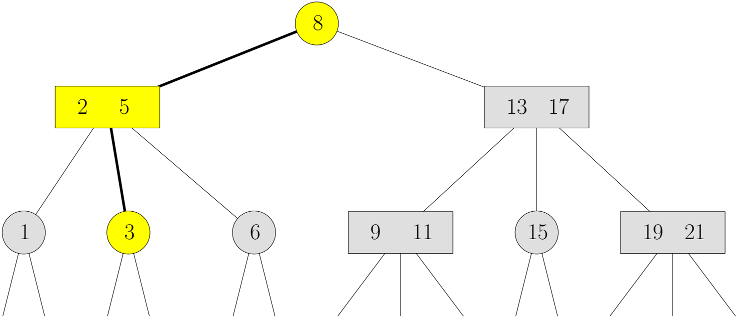
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



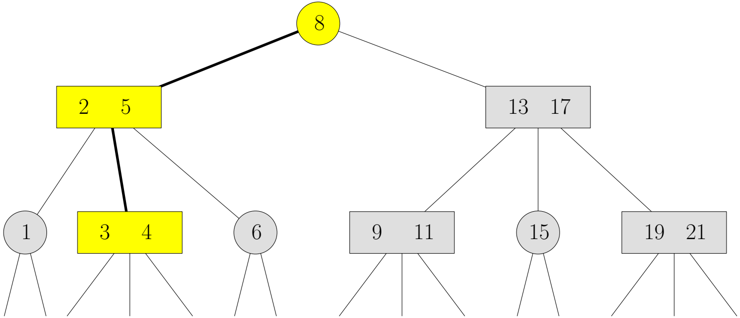
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



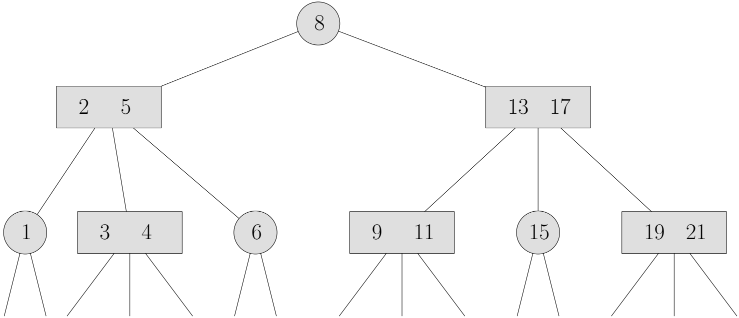
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



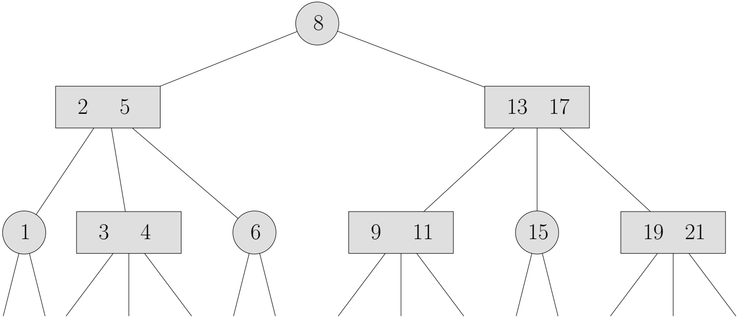
Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



Insertion on a Balanced 2-3-4 Search Tree

Ends at a 4-node (insertion of 4)



There is also the case where the father is also a 4-node.

In this case we need to perform the split again, possible multiple times all the way to the root

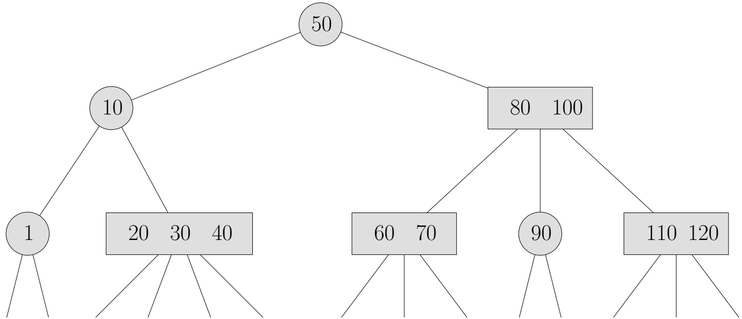
Insertion on a Balanced 2-3-4 Search Tree

In order to simplify insertion and to avoid such cascades, we use the following technique:

- we make sure that the search path does not contain 4-nodes, by spliting any 4-node during our **descend** on the tree

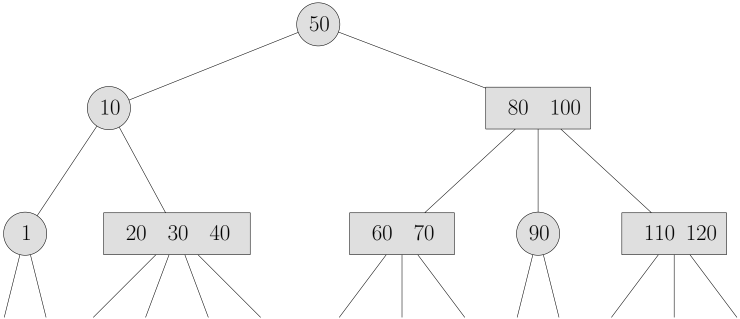
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



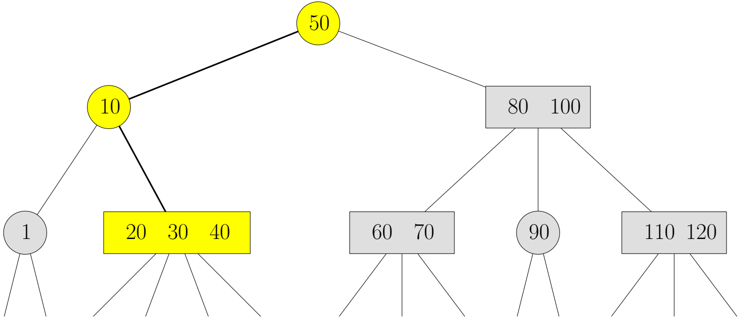
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



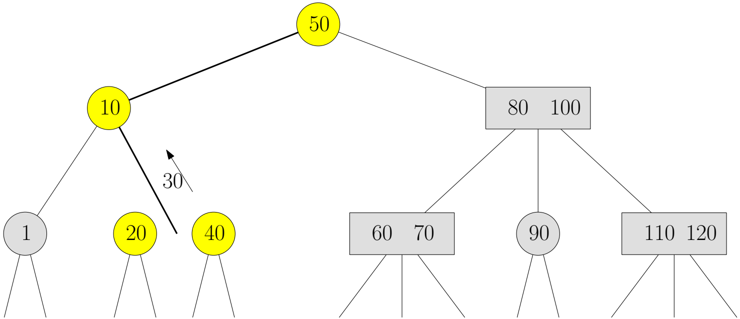
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



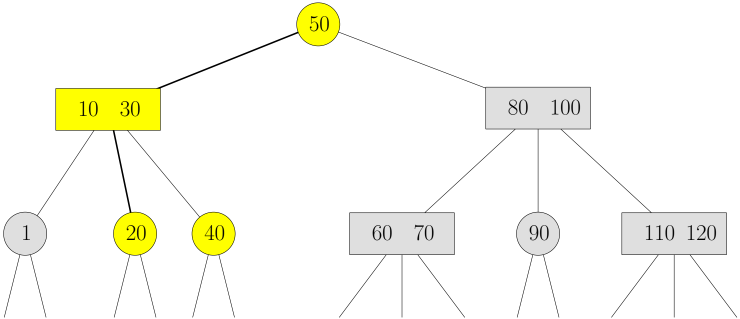
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



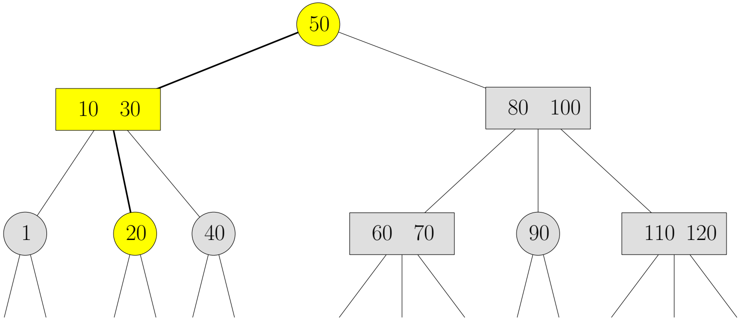
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



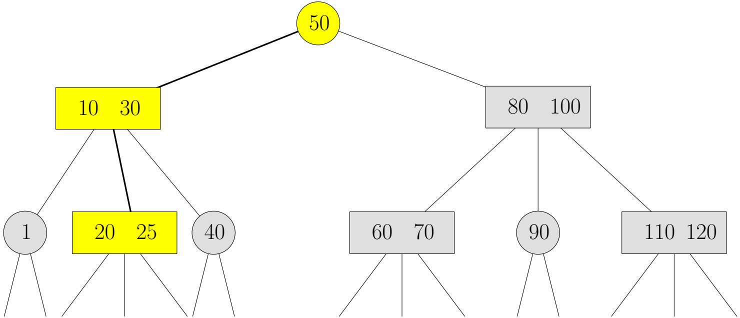
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



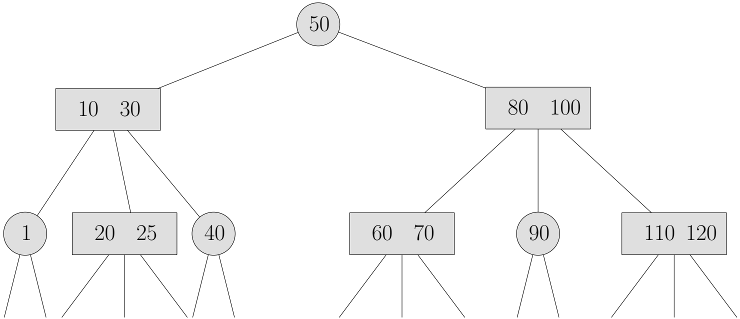
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 25



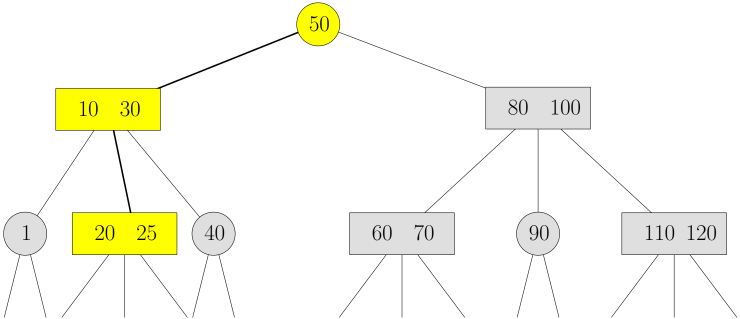
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 29



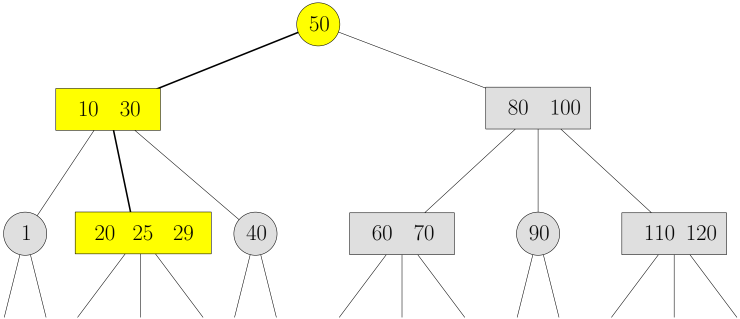
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 29



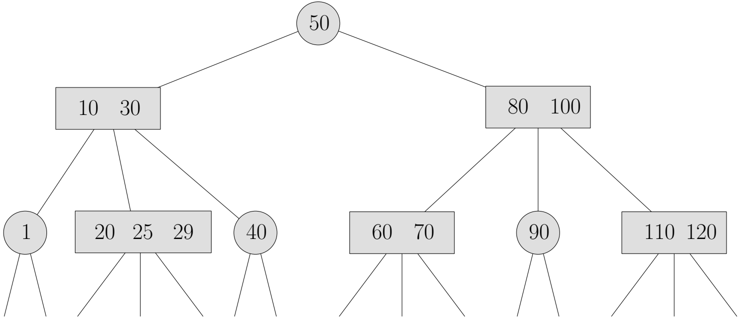
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 29



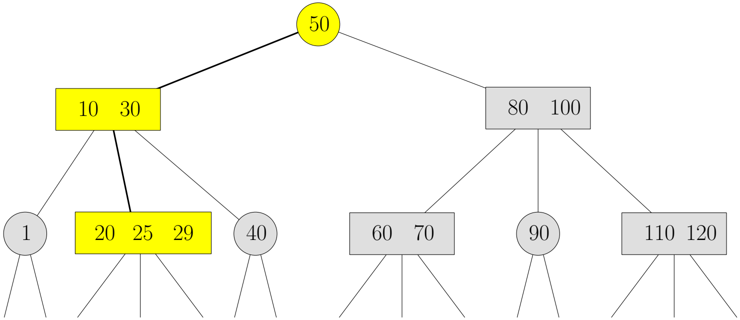
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 29



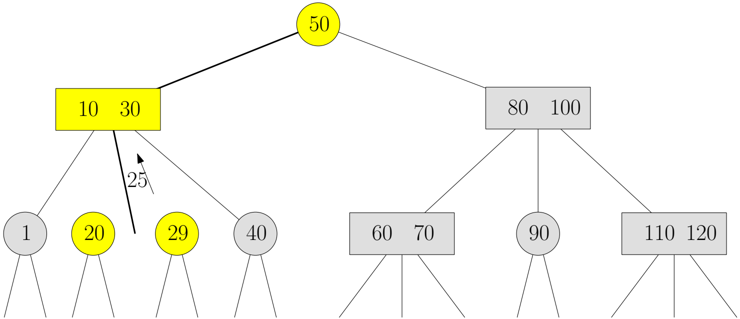
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



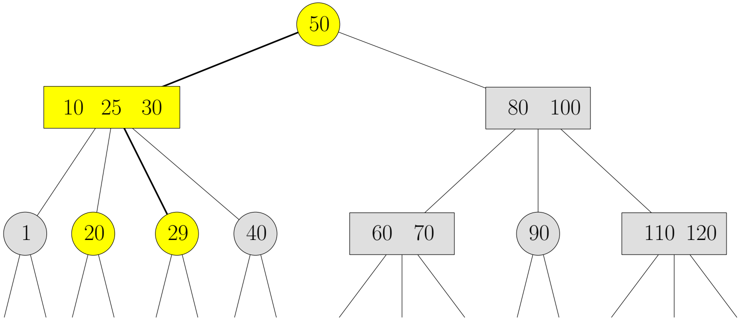
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



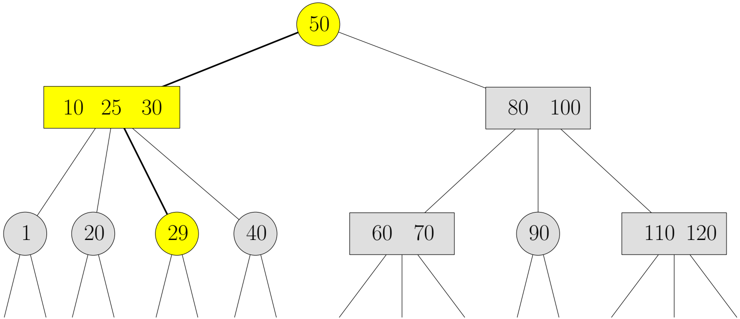
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



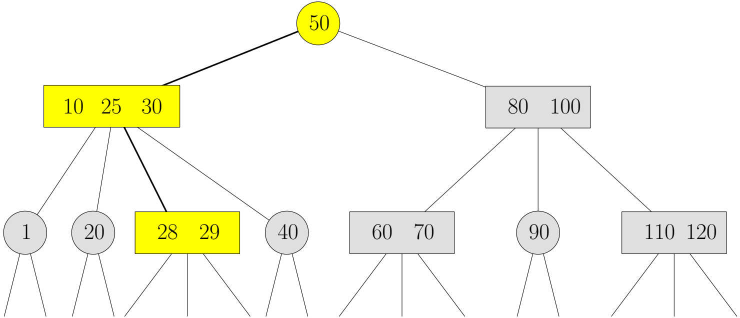
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



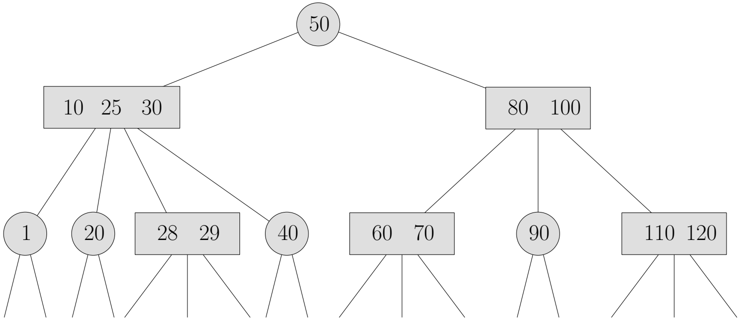
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



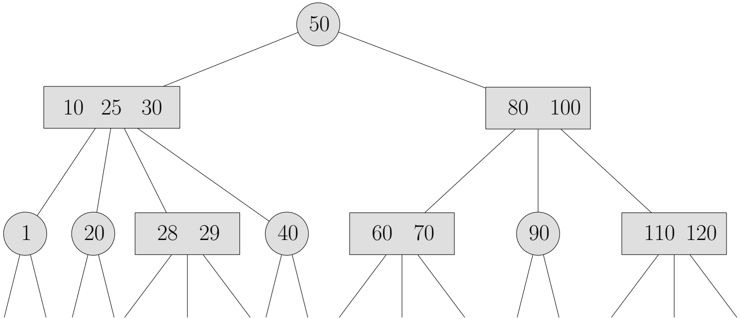
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 28



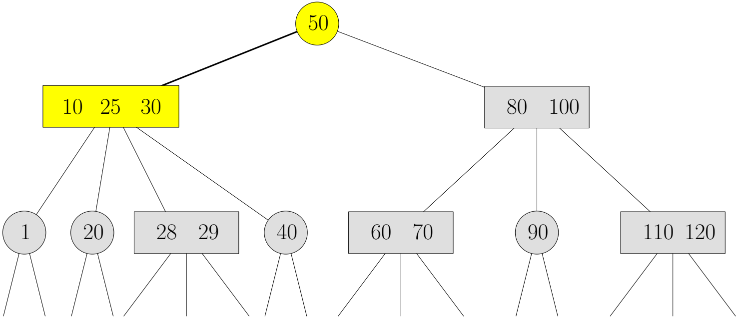
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



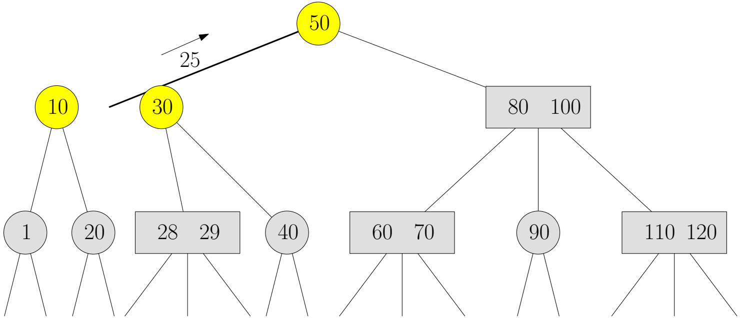
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



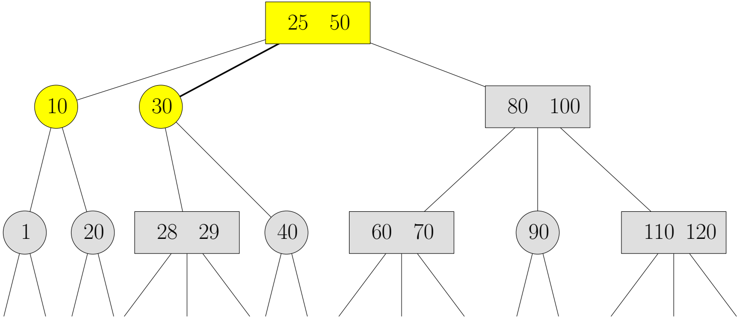
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



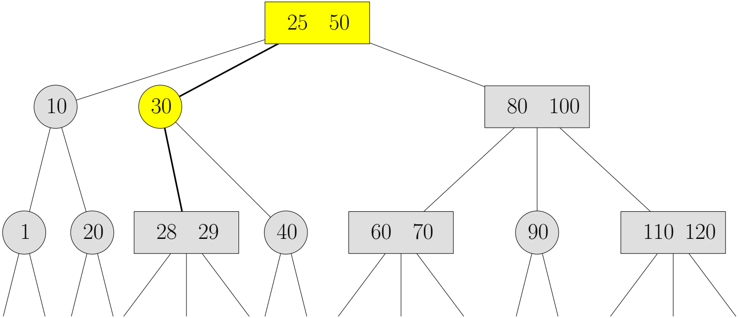
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



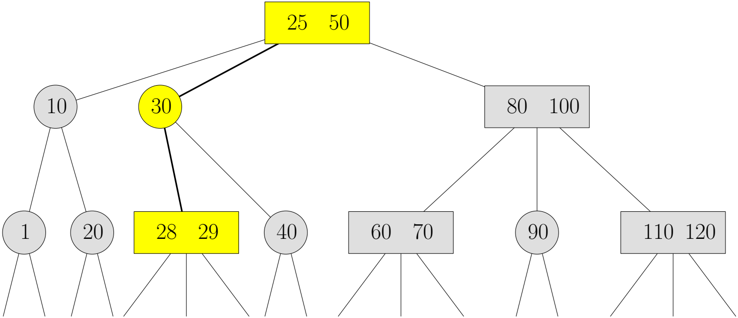
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



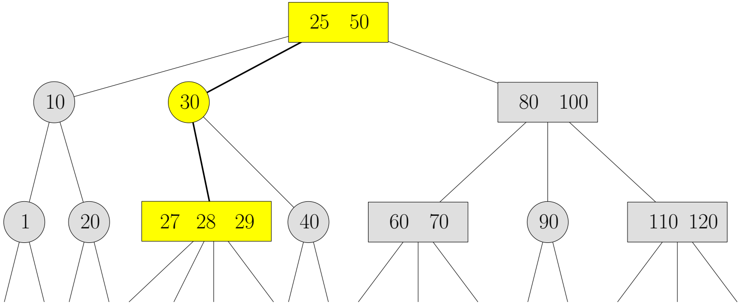
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



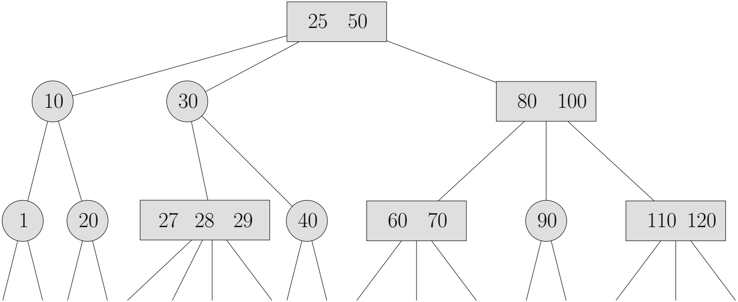
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 27



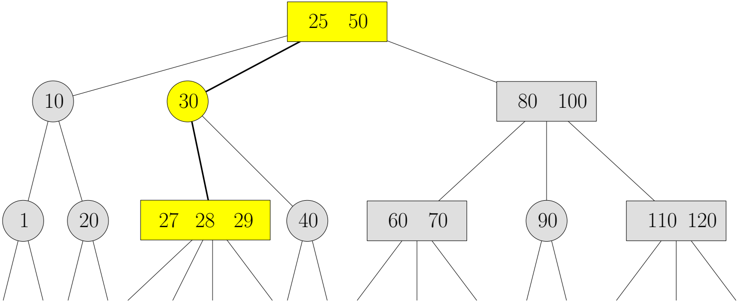
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26



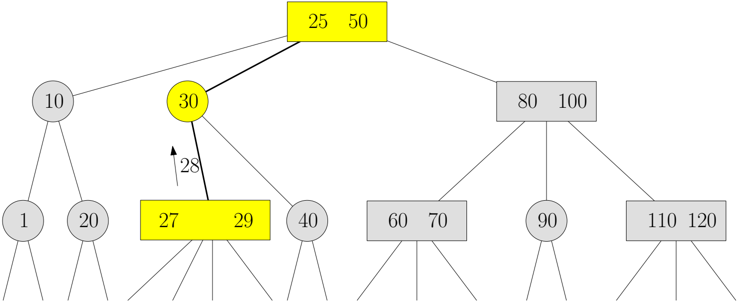
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26



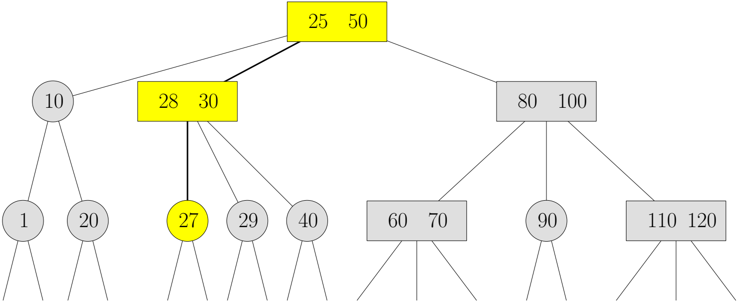
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26



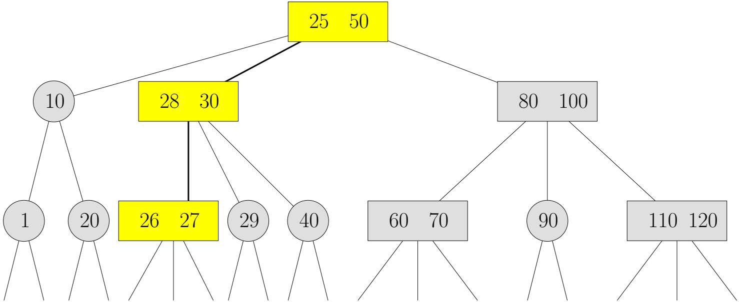
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26



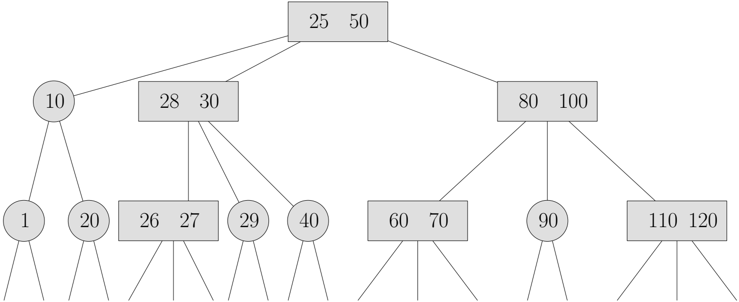
Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26



Insertion on a Balanced 2-3-4 Search Tree

Insertion of 26

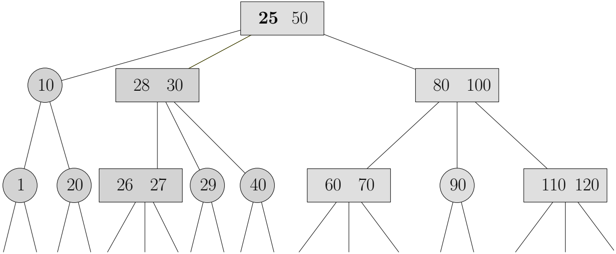


Deletion from a Balanced 2-3-4 Search Tree

Deletion from a non-leaf

- We reduce the problem to the deletion of a leaf
- Same technique as in the BSTs

e.g. to remove 25, we put in its place the *inorder successor (26)* or the *inorder predecessor (20)*.

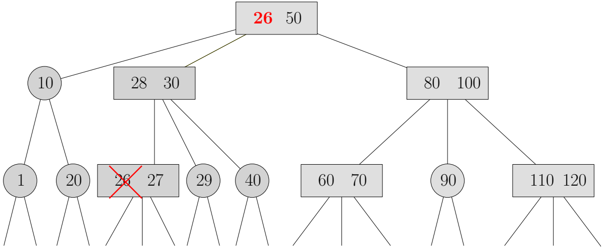


Deletion from a Balanced 2-3-4 Search Tree

Deletion from a non-leaf

- We reduce the problem to the deletion of a leaf
- Same technique as in the BSTs

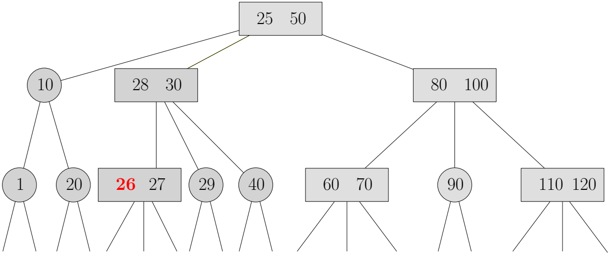
e.g. to remove 25, we put in its place the *inorder successor (26)* or the *inorder predecessor (20)*.



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

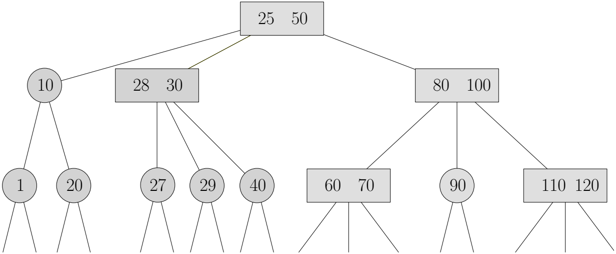
- The leaf has ≥ 2 keys (e.g. delete 26 elow), which means that it is a 3-node or a 4-node.
- Easy case, just delete by switching node types.



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

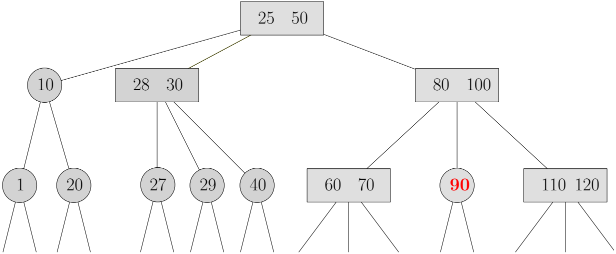
- The leaf has ≥ 2 keys (e.g. delete 26 elow), which means that it is a 3-node or a 4-node.
- Easy case, just delete by switching node types.



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

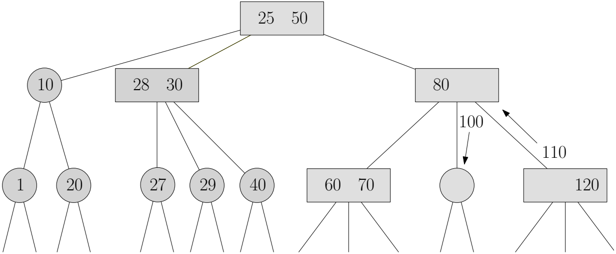
- The leaf has 1 key (underflow), which means it is a 2-node.
- some sibling node has $\geq$ 2 keys
- **balance**: e.g. delete 90



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

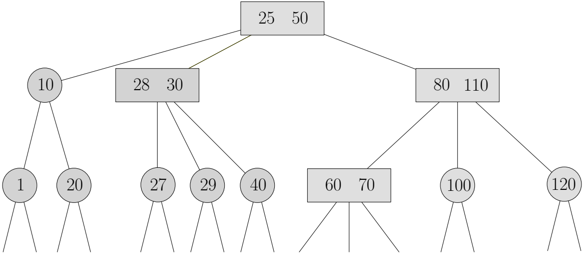
- The leaf has 1 key (underflow), which means it is a 2-node.
- some sibling node has ≥ 2 keys
- **balance**: e.g. delete 90



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

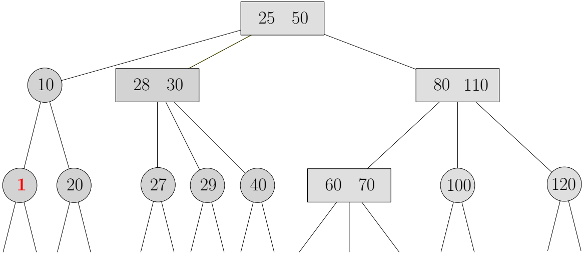
- The leaf has 1 key (underflow), which means it is a 2-node.
- some sibling node has $\geq$ 2 keys
- **balance**: e.g. delete 90



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

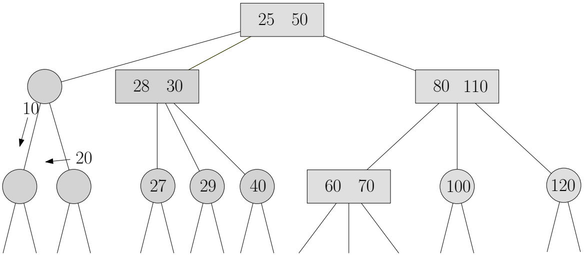
- The leaf has 1 key (underflow), which means it is a 2-node.
- all sibling nodes have 1 key (they are 2-nodes)
- **fusion**: e.g. delete of 1



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

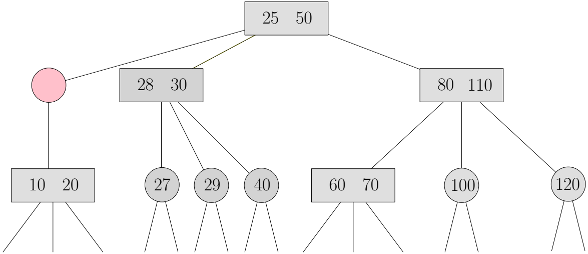
- The leaf has 1 key (underflow), which means it is a 2-node.
- all sibling nodes have 1 key (they are 2-nodes)
- **fusion**: e.g. delete of 1



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

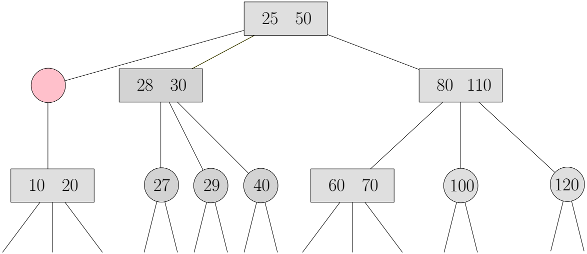
- The leaf has 1 key (underflow), which means it is a 2-node.
- all sibling nodes have 1 key (they are 2-nodes)
- **fusion**: e.g. delete of 1



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

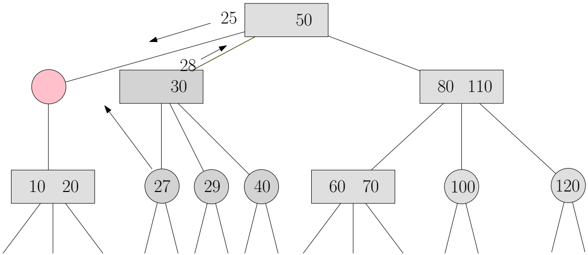
- Now the parent has a problem (underflow)
- repeat the same sequence, either balance or fusion based on the sibling nodes



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

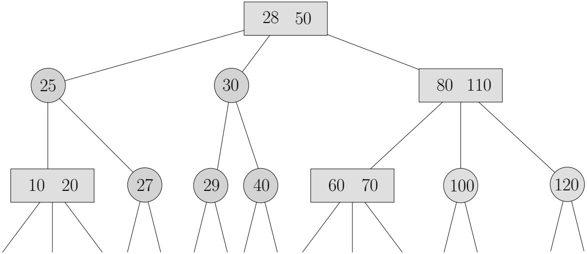
- Now the parent has a problem (underflow)
- repeat the same sequence, either balance or fusion based on the sibling nodes



Deletion from a Balanced 2-3-4 Search Tree

Deletion from a leaf

- Now the parent has a problem (underflow)
- repeat the same sequence, either balance or fusion based on the sibling nodes



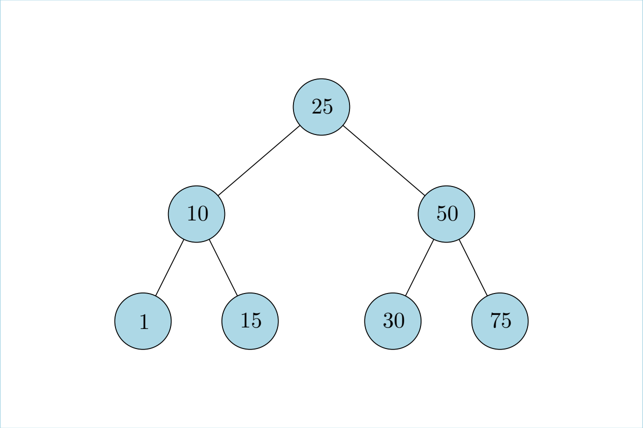
Deletion from a Balanced 2-3-4 Search Tree

- Fusion can cascade up until the root.
- In case it reaches the root, the tree's height is reduced by one.

Deletion costs $\mathcal{O}(\log n)$ time.

Deletion from a Balanced 2-3-4 Search Tree

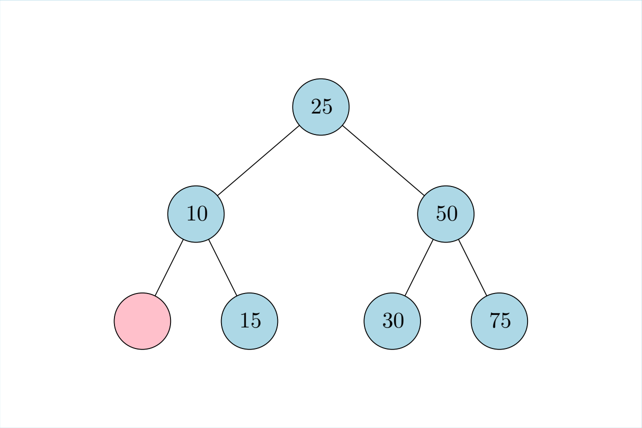
Example of cascade up to the root



Delete key 1

Deletion from a Balanced 2-3-4 Search Tree

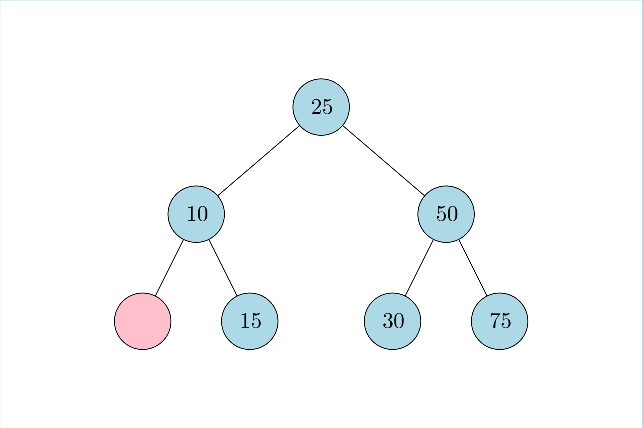
Example of cascade up to the root



Delete key 1

Deletion from a Balanced 2-3-4 Search Tree

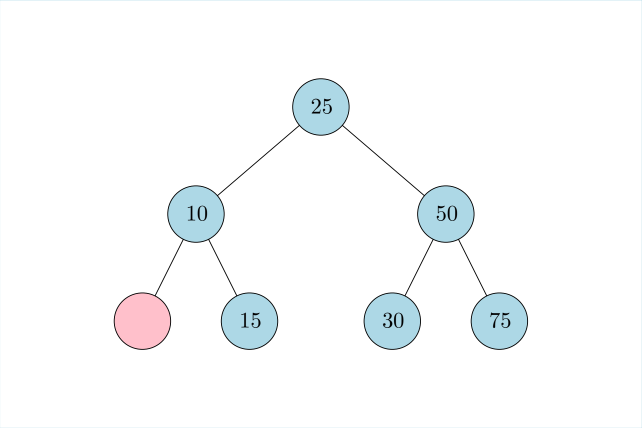
Example of cascade up to the root



Node with underflow - test first balance, then fusion

Deletion from a Balanced 2-3-4 Search Tree

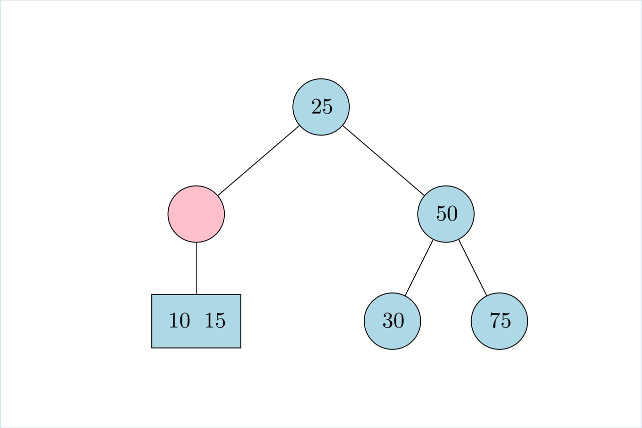
Example of cascade up to the root



Balance not possible - siblings do not have spare keys

Deletion from a Balanced 2-3-4 Search Tree

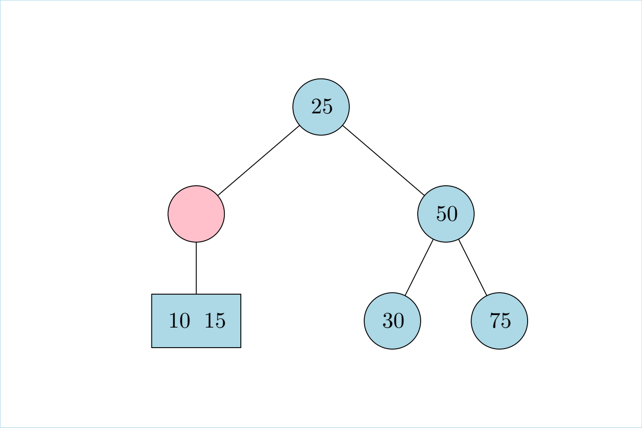
Example of cascade up to the root



Fusion - use both keys from brother and parent

Deletion from a Balanced 2-3-4 Search Tree

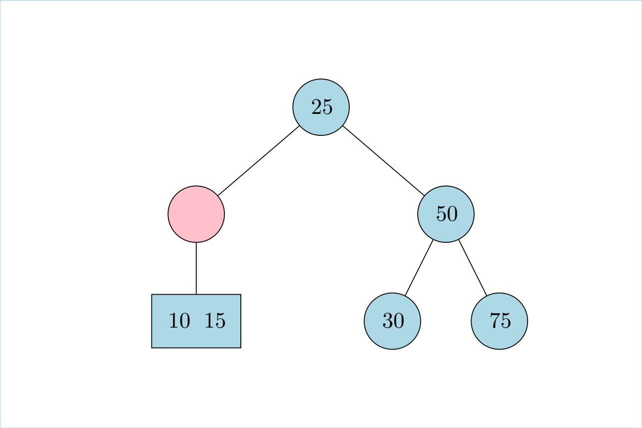
Example of cascade up to the root



Node with underflow - one level up

Deletion from a Balanced 2-3-4 Search Tree

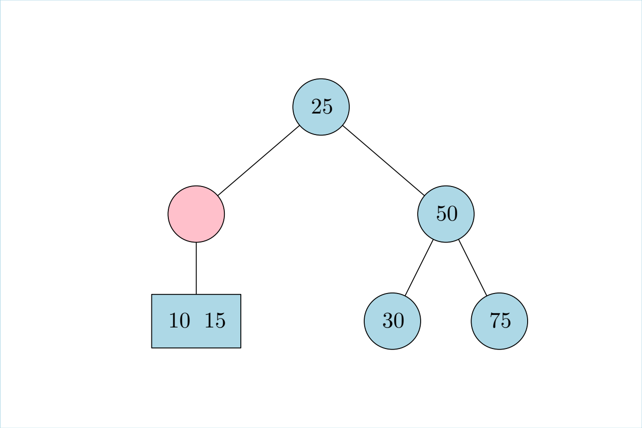
Example of cascade up to the root



Node with underflow - test first balance, then fusion

Deletion from a Balanced 2-3-4 Search Tree

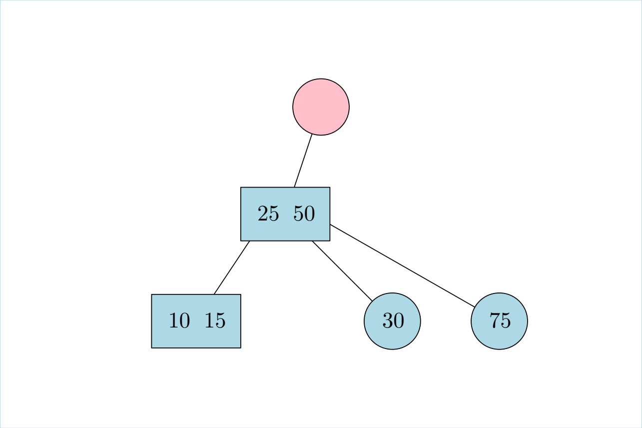
Example of cascade up to the root



Balance not possible - siblings do not have spare keys

Deletion from a Balanced 2-3-4 Search Tree

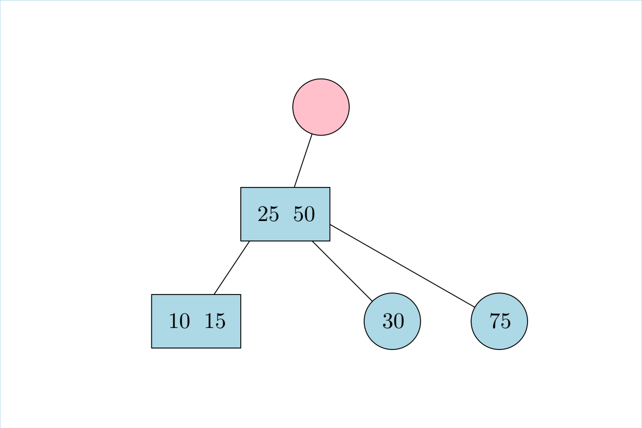
Example of cascade up to the root



Fusion - use both keys from brother and parent

Deletion from a Balanced 2-3-4 Search Tree

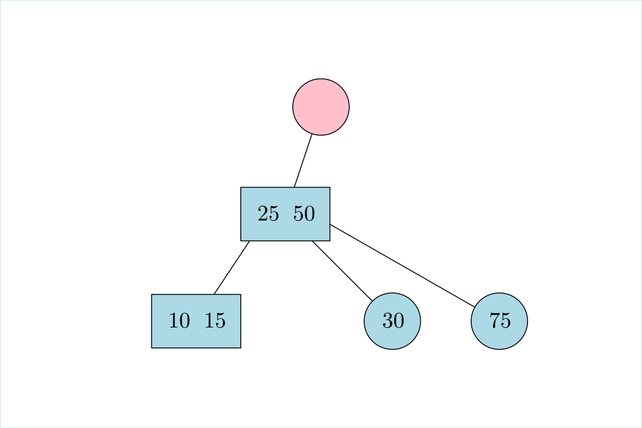
Example of cascade up to the root



Node with underflow - one level up

Deletion from a Balanced 2-3-4 Search Tree

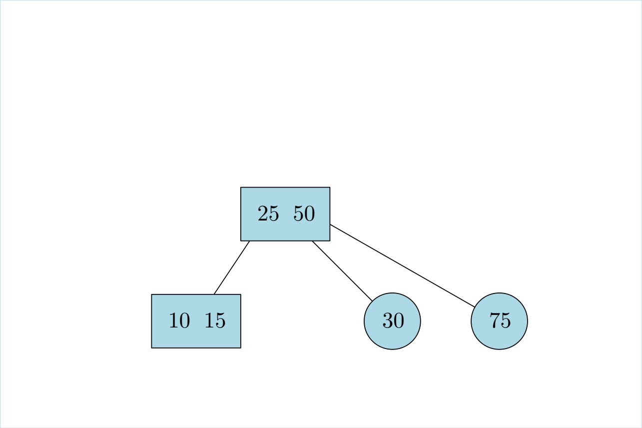
Example of cascade up to the root



If the root has underflow - simply delete it

Deletion from a Balanced 2-3-4 Search Tree

Example of cascade up to the root



If the root has underflow - simply delete it

Balanced 2-3-4 Search Trees

Positive:

- simple insertion algorithm
- complexity $\mathcal{O}(\log n)$ for insertion, deletion and search

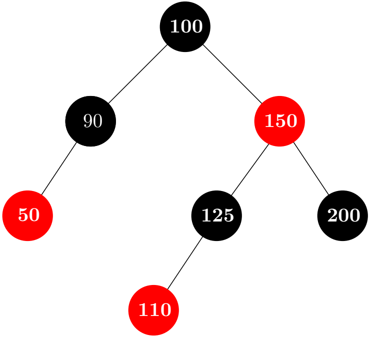
negative:

- 3 different node types
- complex procedured due to multiple links, copying of links, etc.

Red-Black Trees

A Red-Black tree is a binary search tree (BST) with the following additional properties:

- Every node is either red or black
- The root is black
- Every red node does **not** have any red child
- Every path from an external node (null) to the root has to pass through the same number of black nodes



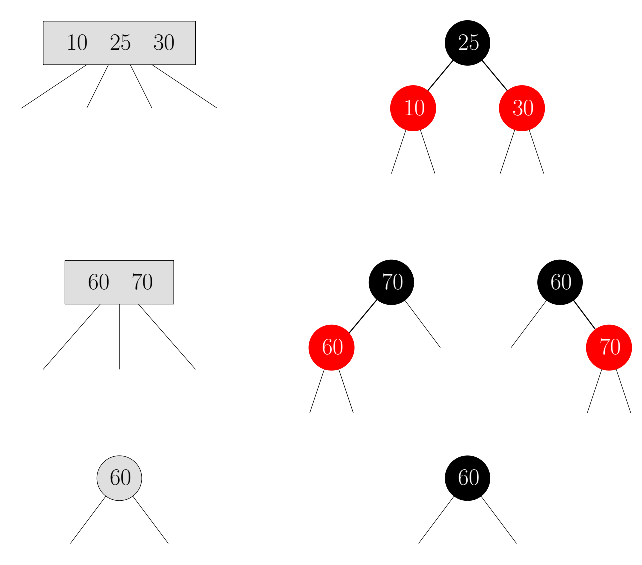
Red-Black Trees

Theorem

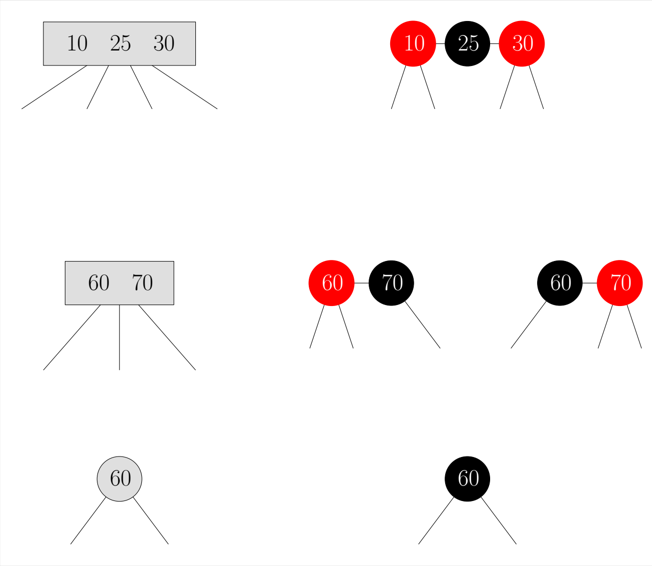
A red-black tree with n nodes has height $\mathcal{O}(\log n)$.

- Instead of a proof we will see a correspondence between read-black trees and 2-3-4 trees.
- We can view red-black trees as a representation of 2-3-4 trees.
- This correspondence allows us to easily show the properties of red-black trees.

Red-Black Trees



Red-Black Trees



Red-Black Trees

Search, insert and delete

Search

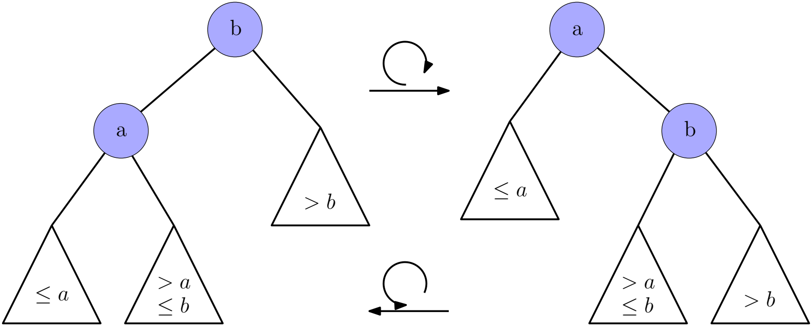
Red-Black keys are binary search trees (BSTs)!

Insertion and Deletion

In correspondence with the 2-3-4 trees, describe the moves by changing colors and rotations.

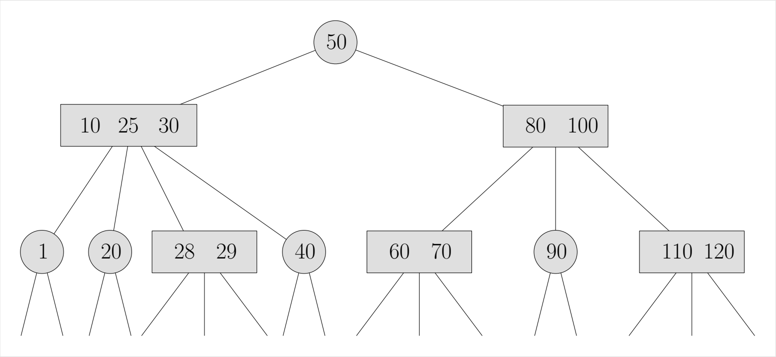
Red-Black Trees

Rotations (on any BST)



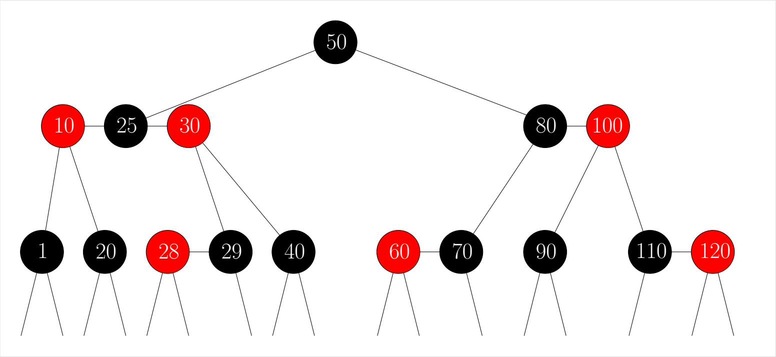
Red-Black Trees

Example



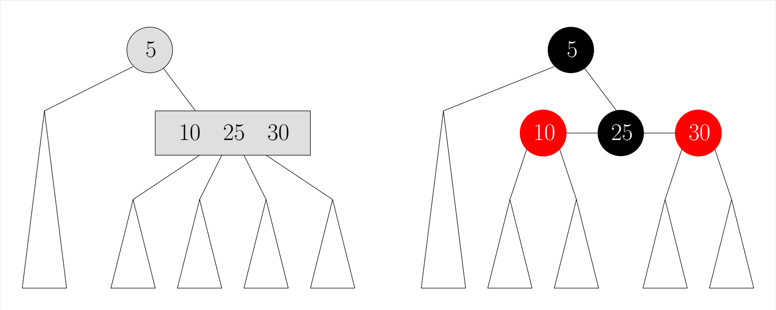
Red-Black Trees

Example



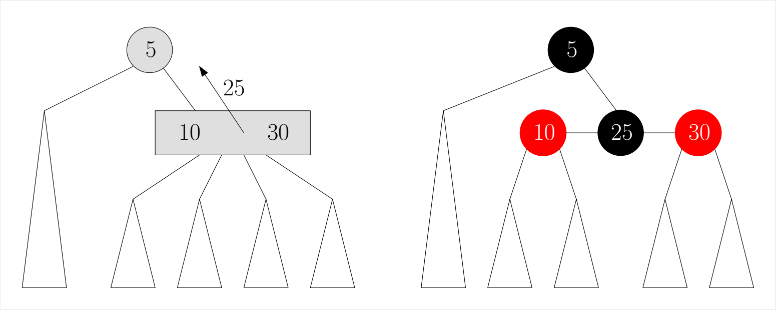
Red-Black Trees

Rotation (first)



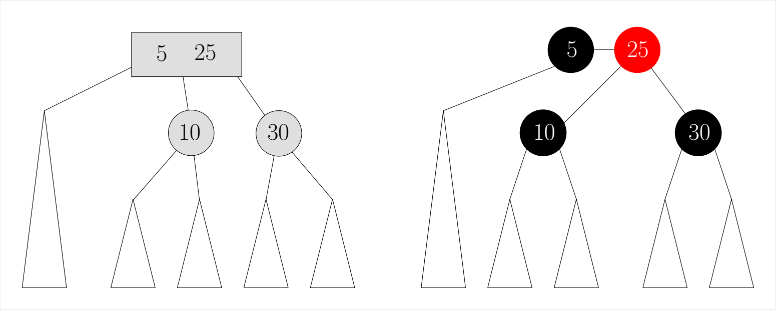
Red-Black Trees

Rotation (first)



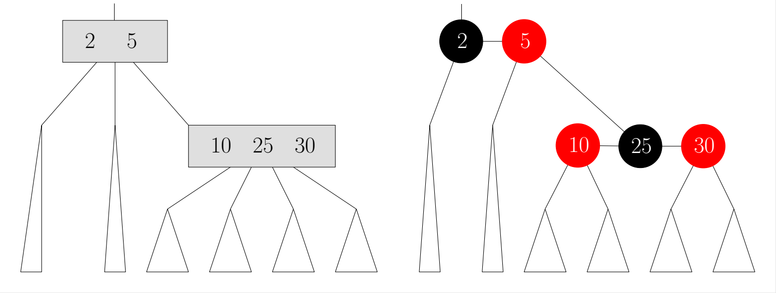
Red-Black Trees

Rotation (first)



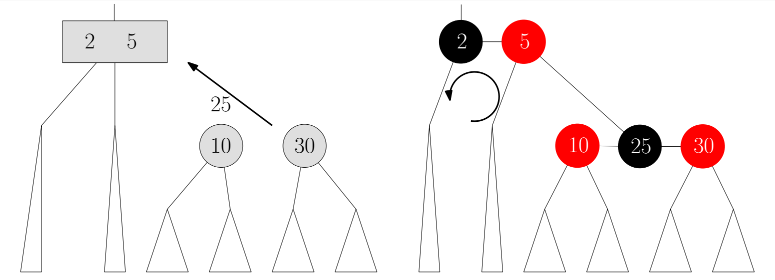
Red-Black Trees

Rotation (second)



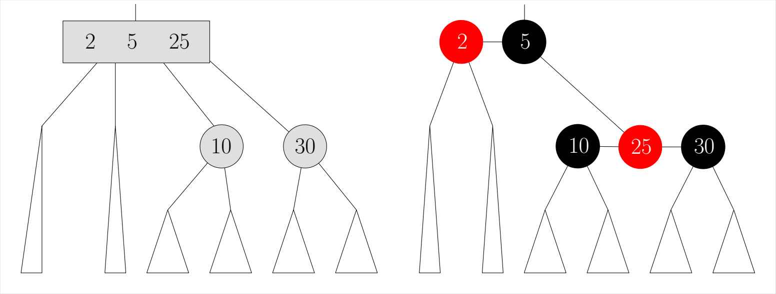
Red-Black Trees

Rotation (second)



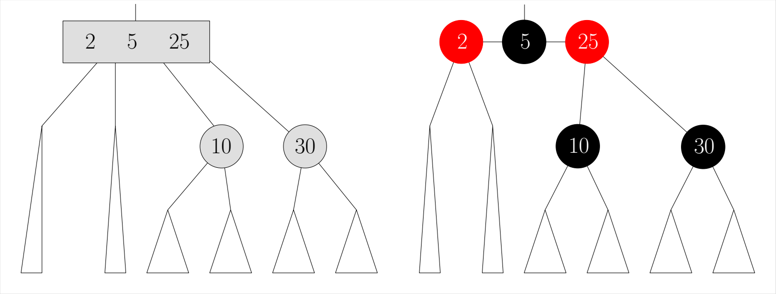
Red-Black Trees

Rotation (second)



Red-Black Trees

Rotation (second)



Red-Black Trees

Rotations

There are more cases...

Red-Black Trees

Insertion and Deletion Complexity

- There are a lot of complicated rules! All of them make sense if we view the corresponding 2-3-4 tree.
- Simulating the changes of a node of a 2-3-4 tree with the corresponding nodes of a red-black tree takes $\mathcal{O}(1)$ time.
- Since 2-3-4 trees support insertion in $\mathcal{O}(\log n)$ time, so do red-black trees.
- The same idea also works for deletions.

Red-Black Trees

Height

Theorem

A red-black tree with n nodes has height $\mathcal{O}(\log n)$.

Proof.

We can collect all red nodes to their parents in order to convert the tree to a 2-3-4 tree.

The height of the tree can at most lose half of its size.

The resulting 2-3-4 tree has height $\mathcal{O}(\log n)$, thus the original red-black tree has height at most $2 \cdot \mathcal{O}(\log n) = \mathcal{O}(\log n)$.

☐

Other Balanced Trees

- AVL trees: $\mathcal{O}(\log n)$ height using rotations
- splay trees
- scapegoat trees
- etc.

AVL Trees

The first balanced tree!

G.M. **A**delson-**V**elskii and E.M. **L**andis. An algorithm for the organization of information, Proceedings of the USSR Academy of Sciences 146: 263–266, 1962.

Named from the initials of its inventors.

AVL Trees

- Binary Search Tree
- Additional balance property

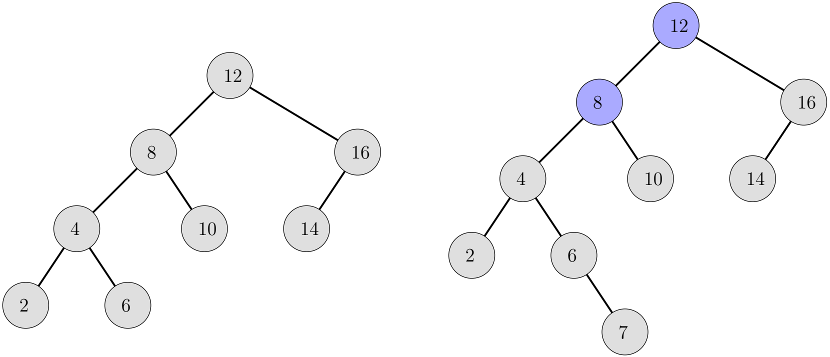
The additional balance property needs to be maintained easily and to be able to preserve the height of the tree to $\mathcal{O}(\log n)$.

AVL Trees

Definition

An AVL tree is a binary search tree T where for each node $v \in T$ the height of its two subtrees differ by at most 1. The height of an empty tree is defined as -1.

AVL Trees



Left AVL tree, right non-AVL tree.

AVL Trees

Height

Theorem

An AVL tree with n keys has height $\mathcal{O}(\log n)$.

AVL Trees

Height

Theorem

An AVL tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let $n(h)$ be the smallest number of nodes of an AVL tree with height h.

□

AVL Trees

Height

Theorem

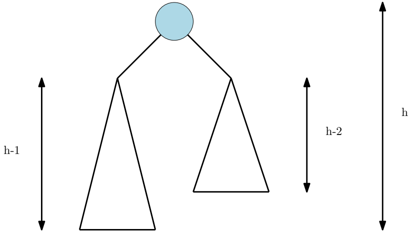
An AVL tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let $n(h)$ be the smallest number of nodes of an AVL tree with height h.

We can easily show that $n(1) = 1$ and $n(2) = 2$.

For $h > 2$ an AVL tree has a root and two subtrees, one AVL subtree with height $h - 1$ and one AVL subtree with height $h - 2$.



Thus $n(h) = 1 + n(h-1) + n(h-2)$.

☐

AVL Trees

Height

Theorem

An AVL tree with n keys has height $\mathcal{O}(\log n)$.

Proof.

Let $n(h)$ be the smallest number of nodes of an AVL tree with height h.

Since $n(h-1) > n(h-2)$ we get that

$$n(h) = 1 + n(h-1) + n(h-2) > 2n(h-2) > 4n(h-4) > 8n(h-6) > \dots$$

using induction

$$n(h) > 2^i n(h-2i)$$

Since $n(1) = 1$ we have that $n(h) > 2^{(h-1)/2}$ and by taking logarithms we have $h < 2 \log n(h) + 1$.

□

Insertion on an AVL tree

Insertion is performed like in any other BST but afterwards we might need to fix the tree.

We make sure we perform $\mathcal{O}(d)$ operations where d is the depth where insertion is performed.

Insertion on an AVL tree

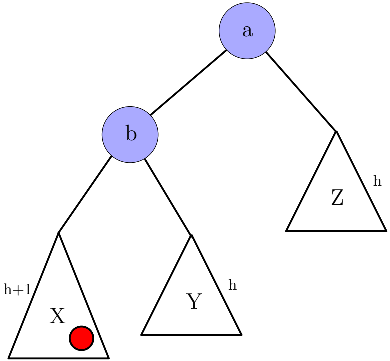
We fix the tree with rotations. Let x be the node with the largest depth that is non-balanced.

There are 4 cases based on whether the insertion happened at

1. the left subtree of the left child of x
2. the right subtree of the left child of x
3. the left subtree of the right child of x
4. the right subtree of the right child of x

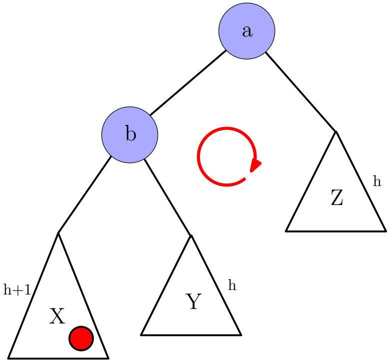
Cases 1&4 are resolved with a *single rotation* while cases 2&3 needs a *double rotation*.

Single Rotation



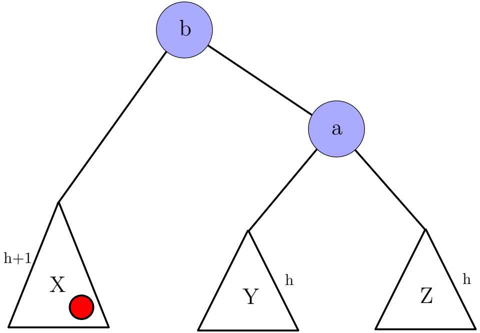
$X < b < Y < a < Z$

Single Rotation



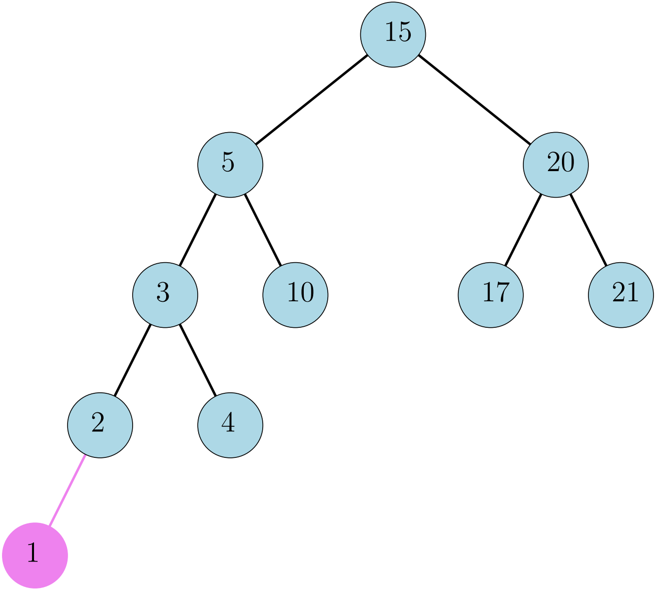
$$X < b < Y < a < Z$$

Single Rotation

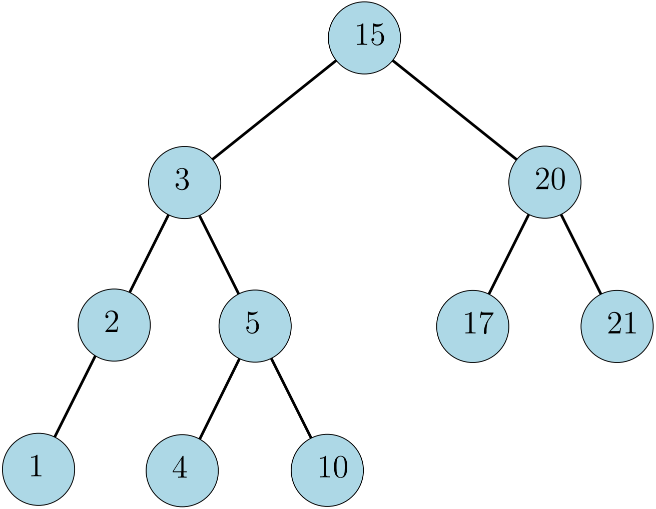


$$X < b < Y < a < Z$$

Single Rotation

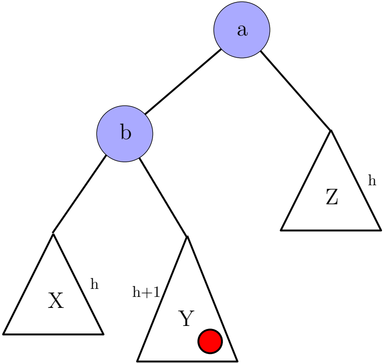


Single Rotation



Other cases

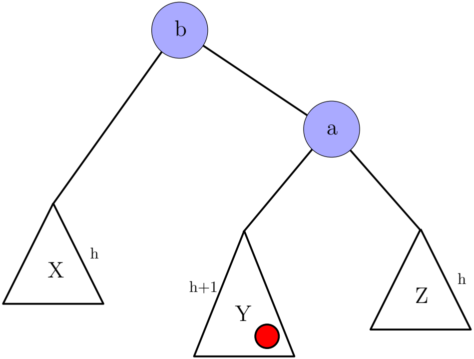
In cases 2&3 the single rotation does not work.



$$X < b < Y < a < Z$$

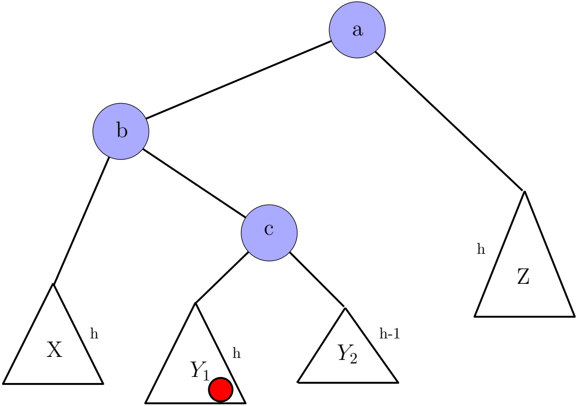
Other cases

In cases 2&3 the single rotation does not work.



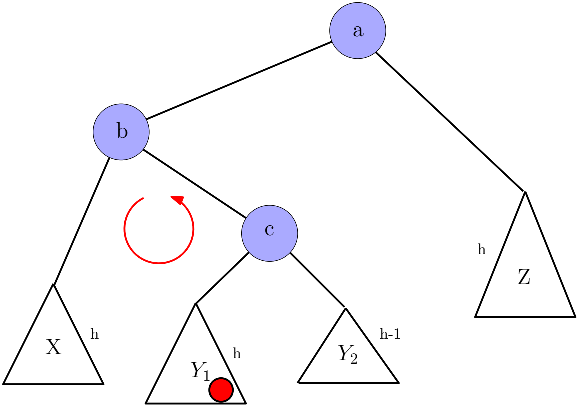
$$X < b < Y < a < Z$$

Double Rotation



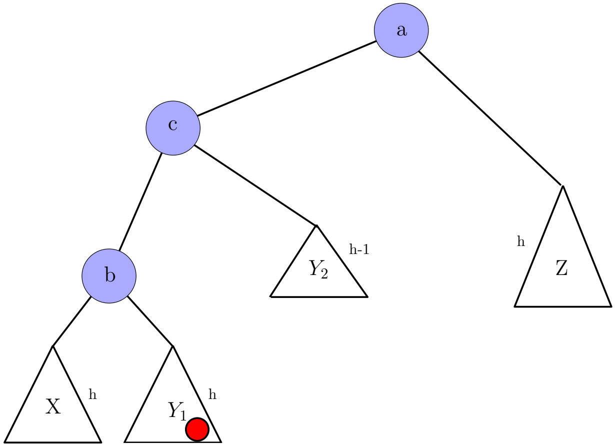
$$X < b < Y_1 < c < Y_2 < a < Z$$

Double Rotation



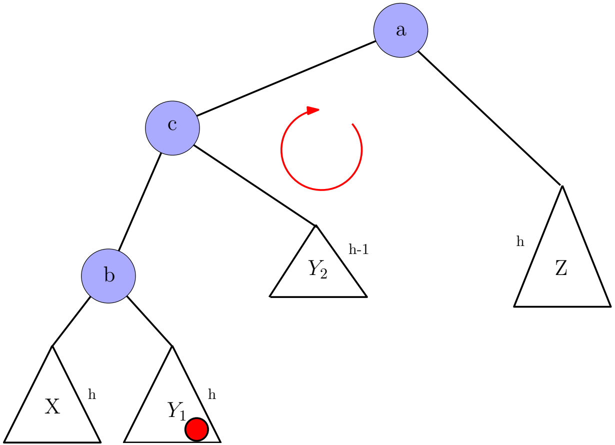
$$X < b < Y_1 < c < Y_2 < a < Z$$

Double Rotation



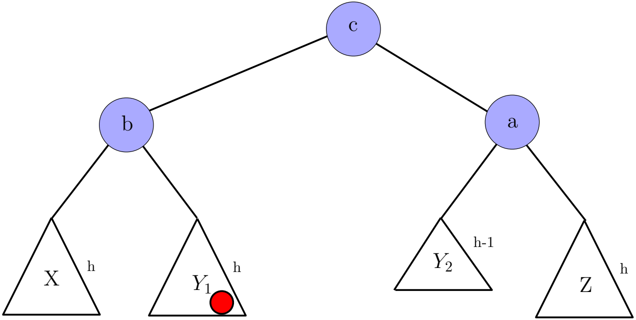
$$X < b < Y_1 < c < Y_2 < a < Z$$

Double Rotation



$$X < b < Y_1 < c < Y_2 < a < Z$$

Double Rotation



$$X < b < Y_1 < c < Y_2 < a < Z$$

Deletion

Deleting elements from an AVL tree happens in the same way as in BSTs with additional rotations.

$\mathcal{O}(\log n)$ in the worst case

Memory Hierarchy

(Year 2010)

A big dilemma exists between speed and size when it comes to memory.

SRAM (Static Random Access Memory)

- Registers are made using SRAM
- Very fast ($1ns = 10^{-9}sec$ - handles GHz clock speeds)
- Very expensive ($1GB \approx 5000\$$)

DRAM (Dynamic Random Access Memory)

- Memory is made using this technology
- Fast (25ns)
- Less expensive ($16GB \approx 100\$$)

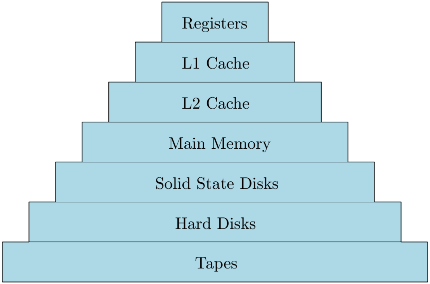
Hard Disk

- Very slow ($5ms = 5 \cdot 10^{-3}sec$)
- Very cheap

Memory Hierarchy

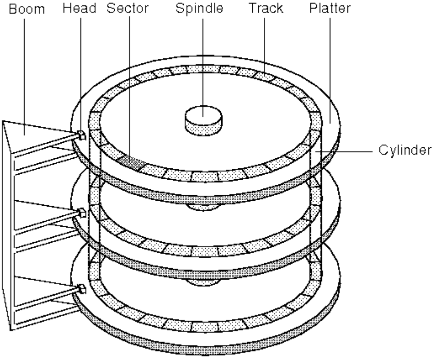
(Year 2010)

Idea: Use many different types of memory for better performance.



Registers	0.25 - 1ns
L1 Cache	1 - 5ns
L2 Cache	5 - 25ns
Main Memory	25 -100ns
Solid State Disks	100 - 150ns
Hard Disks	2 - 20ms
Tapes	sequential access

External Memory



- seek time: time in order for the heads to move to the right location (track)
- rotational delay: waiting time in order for the cylinder to come to the right sector that we are looking for

External Memory

- computer memory (DRAM) has access time of nanoseconds: 10^{-9} sec, e.g. 25 - 100 nsec
- hard disks have access time of milliseconds: 10^{-3}, e.g. 2 - 20 ms

Conclusion

It is not a good idea to read small amounts of data from the disk. For this reason disks always read/write a block which is in the order of Kilobytes, e.g. 512K.

B-Tree (Beta Tree)

- search tree which is efficient for storing to disk
- generalizes the 2-3-4 trees using nodes with number of keys between t and $2t$ for any $t \geq 2$.

B-tree Definition

A *B-tree* T is a tree with a root and the following properties:

1. every node x has the following fields:
   (a) $n[x]$, the number of keys that x has
   (b) the $n[x]$ keys in order: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
   (c) a value $leaf[x]$ which is true iff node x is a leaf

B-tree Definition

A *B-tree* T is a tree with a root and the following properties:

1. every node x has the following fields:
   (a) $n[x]$, the number of keys that x has
   (b) the $n[x]$ keys in order: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
   (c) a value $leaf[x]$ which is true iff node x is a leaf
2. if x is an internal node it also contains $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leafs do not have children in which case these pointers are not defined.

B-tree Definition

A *B-tree* T is a tree with a root and the following properties:

1. every node x has the following fields:
   (a) $n[x]$, the number of keys that x has
   (b) the $n[x]$ keys in order: $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$
   (c) a value $leaf[x]$ which is true iff node x is a leaf
2. if x is an internal node it also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leafs do not have children in which case these pointers are not defined.
3. The keys $key_i[x]$ split the range of the keys which are stored in each subtree: if k_i is the key stored in subtree with root $c_i[x]$

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \cdots \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$

B-tree Definition

4. Every leaf has the same depth, the height of the tree h.

B-tree Definition

4. Every leaf has the same depth, the height of the tree h.
5. Let $t \geq 2$ be the **minimum order** of the tree:
    5.1 Every node besides the root must have at least $t-1$ keys (which translates to at least t children). If the tree is not null, the root must have at least one key.
    5.2 Every node can contain at most $2t-1$ keys, e.g. at most $2t$ children. A node is called **full** if it contains exactly $2t-1$ keys.

B-tree Definition

4. Every leaf has the same depth, the height of the tree h.
5. Let $t \geq 2$ be the **minimum order** of the tree:
    5.1 Every node besides the root must have at least $t - 1$ keys (which translates to at least t children). If the tree is not null, the root must have at least one key.
    5.2 Every node can contain at most $2t - 1$ keys, e.g. at most $2t$ children. A node is called **full** if it contains exactly $2t - 1$ keys.

The simplest B-tree is for $t = 2$. Every internal node has 2, 3 or 4 children, thus, is the 2-3-4 tree.

In practice we use much larger values for t.

B-tree Height

Theorem

A B-tree with $n \geq 1$ keys and minimum order $t \geq 2$ has height

$$h \leq log_t \frac{n+1}{2}.$$

Proof.

A B-tree with height h has the least number of nodes if the root has one key and the rest have $t-1$ keys.

At lever 0 there is one node. At level 1 there are 2 nodes. At level 2 there are $2t$ nodes, at level 3 $2t^2$ nodes, etc. ☐

B-tree Height

Theorem

A B-tree with $n \geq 1$ keys and minimum order $t \geq 2$ has height

$$h \leq log_t \frac{n+1}{2}.$$

Proof.

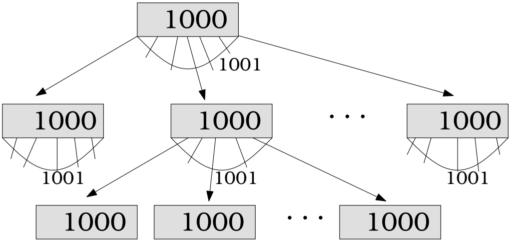
The number of keys are:

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \end{aligned}$$

□

Disk Accesses

Making sure that each node fits exactly on a disk block, we can use disk accesses equal to the height of the tree.



1 κόμβος
1000 κλειδιά

1001 κόμβοι
1.001.000 κλειδιά

1.002.001 κόμβοι
1.002.001.000 κλειδιά

The tree has height $\mathcal{O}(\log_t n)$.

Basic Principles when Implementing B-trees

(i) The root of a b-tree always remains in memory

(ii) We need two functions which read and write nodes to disk

  - `DISK-READ(x)`
  - `DISK-WRITE(x)`

Search on a B-tree

Search works just like in BSTs except now we have more choices per node.

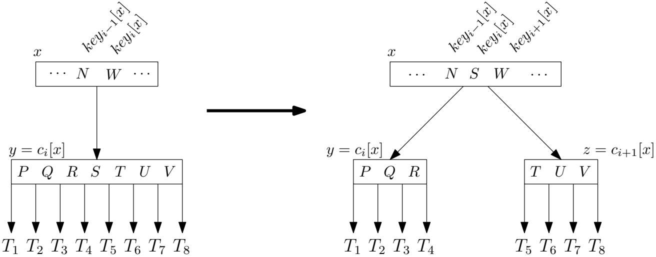
```
$B - TREE - SEARCH(x, k)$

$i = 1$
while $i \leq n[x]$ and $k > key_i[x]$ do
    $i = i + 1$
end
if $i \leq n[x]$ and $k = key_i[x]$ then
    return $(x, i)$
end
if $leaf[x]$ then
    return $nil$
else
    $DISK - READ(c_i[x])$
end
return $B - TREE - SEARCH(c_i[x], k)$
```

Insertion in a B-tree

- While searching for the insertion location, we split full nodes (having $2t - 1$ keys) into two nodes
- When a node splits, the middle key goes up the tree.
- This makes enough room for possible splits at lower levels of the tree.

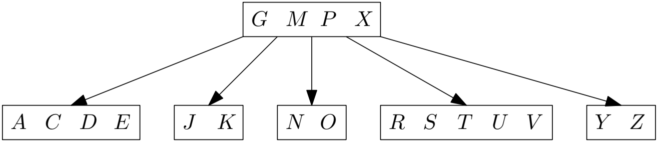
Node split

$t = 4$



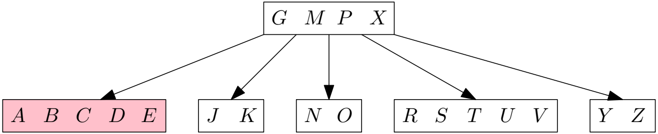
Example

$t = 3$



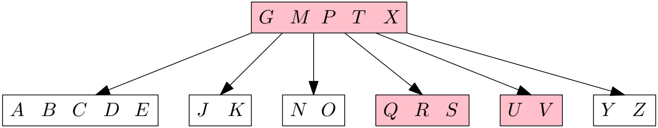
Example

$t = 3$, insertion of B



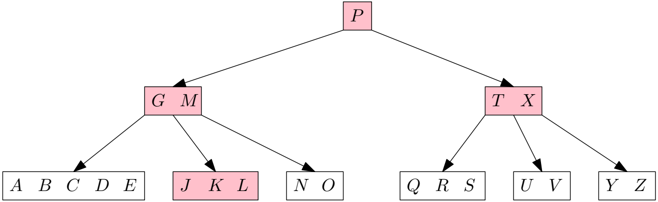
Example

$t = 3$, insertion of Q



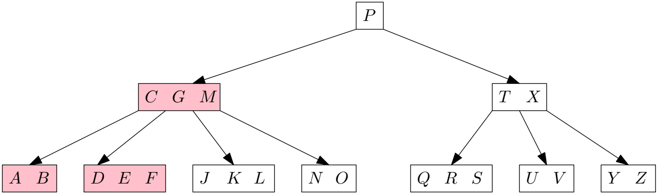
Example

$t = 3$, insertion of L



Example

$t = 3$, insertion of F



Deletion from a B-tree

Works the same as in 2-3-4 trees.

We may need to either

- balance, or
- fuse

Reading and Sources

- Sections 7.1, 7.2, 7.3

  Kurt Mehlhorn Peter Sanders. Algorithms and Data Structures, The basic toolbox, 1/e, 2014.
- Sections 12.5, 12.6, 12.8, 12.9, 13.3, 13.4, 16.3

  Robert Sedgewick. Algorithms in C: Fundamentals, Data Structures, Sorting, Searching. 3/e, Addison-Wesley Professional, 1997.
- Sections 12.1, 12.2, 12.3, 13.1, 13.2, 13.3, 13.4

  Cormen, Leiserson, Rivest and Stein. Introduction to Algorithms. 3/e, MIT Press. 2009.