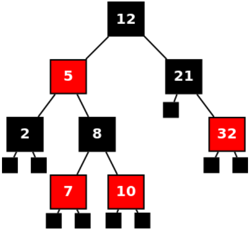
Balanced Binary Search Trees

Pedro Ribeiro

DCC/FCUP

2021/2022



Motivation

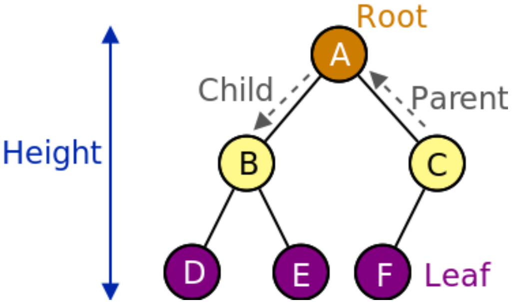
- Let S be a set of **"comparable"** objects/items:
  - Let a and b be two different objects. They are "comparable" if it is possible to say that $a < b$, $a = b$ or $a > b$.
  - Example: numbers, but we could have other data types (students with names and numbers, teams with points and goal-average, ...)

- A few possible **problems** of interest:
  - Given a set S, determine if **a certain item is in** S
  - Given a **dynamic** set S (that changes with insertions and removals), determine if **a certain item is in** S
  - Given a **dynamic** set S, determine the **min/max** item in S
  - Given a **dynamic** set S, determine the elements in a range $[a, b]$
  - **Sort** a set S
  - ...

- **Binary Search Trees!**

Binary Trees - Notation

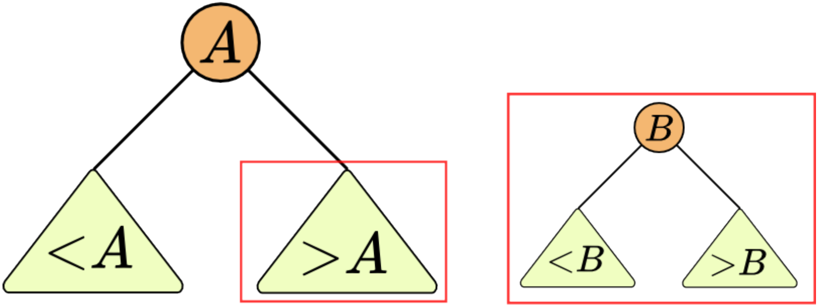
- An overview of **notation** for binary trees:



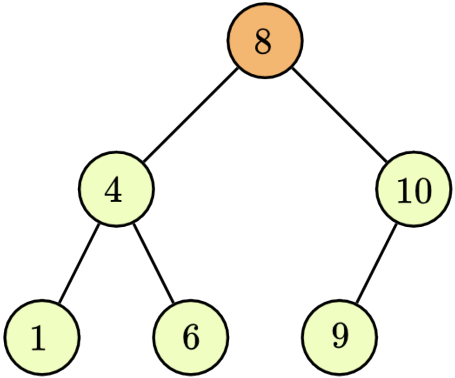
- Node A is the **root** and nodes D, E and F are the **leafs**
- Nodes $\{B, D, E\}$ are a **subtree**
- Node A is the **parent** of nodes B and C
- Nodes D and E are **children** of node B
- Node B is a **brother** of node C
- ...

Binary Search Trees - Overview

- For **all** nodes of tree, the following must hold:
  the node is bigger than all nodes in the left subtree and smaller than all nodes in the right subtree

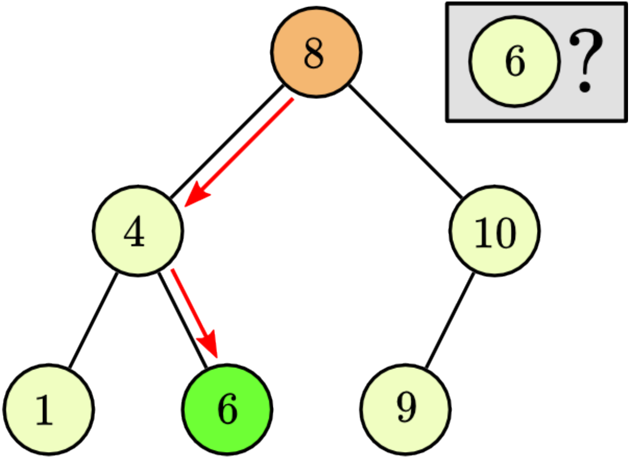


Binary Search Trees - Example



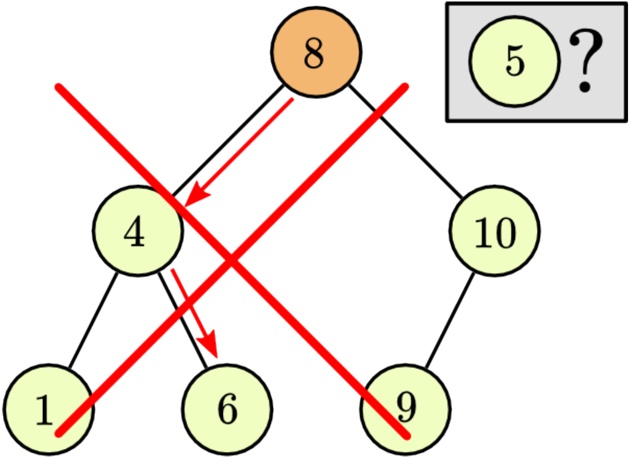
- The **smallest** element is... in the **leftmost node**
- The **biggest** element is... in the **rightmost node**

- **Searching for values** in binary search trees:



Binary Search Trees - Search

- **Searching for values** in binary search trees:



Binary Search Trees - Search

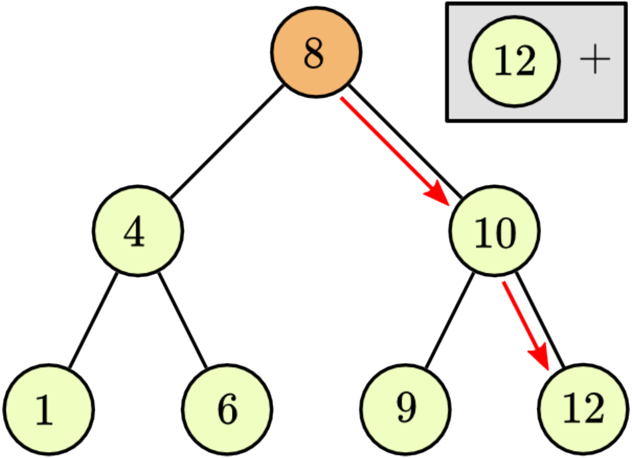
- **Seaching for values** in binary search trees:

Searching in a binary search tree (true/false to check if exists)

Search(T, v**)**:
  If Null(T**) then**
    return *false*
  Else If $v < T.value$ **then**
    return Search($T.left\_child, v$**)**
  Else If $v > T.value$ **then**
    return Search($T.right\_child, v$**)**
  Else
    return *true*

Binary Search Trees - Insertion

- **Inserting values** in binary search trees:



Binary Search Trees - Insertion

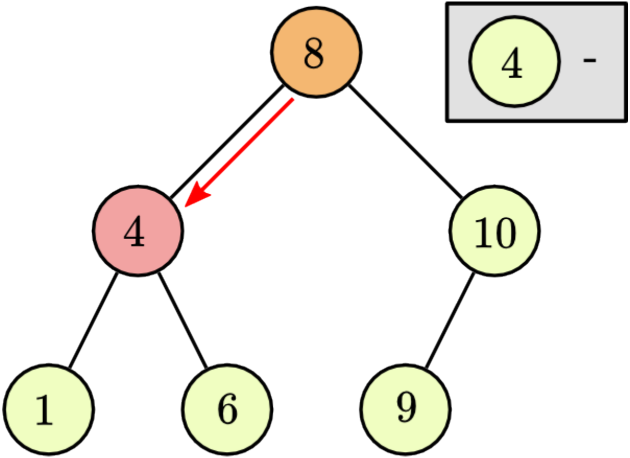
- **Inserting values** in binary search trees:

Insertion on a binary search tree

Insert(T, v):
  If Null(T) **then return new Node**(v)
  If $v < T.value$ **then**
    $T.left\_child =$ **Insert**($T.left\_child, v$)
  Else If $v > T.value$ **then**
    $T.right\_child =$ **Insert**($T.right\_child, v$)
  return T

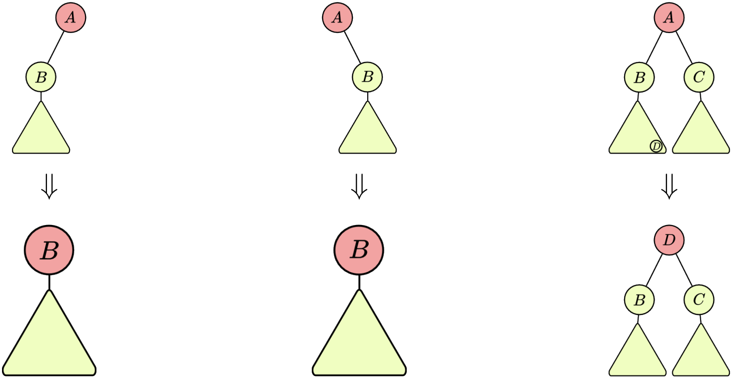
Binary Search Trees - Removal

- **Removing values** from binary search trees:



Binary Search Trees - Removal

- After finding the node we need to decide **how to remove**
  - 3 possible cases:



Binary Search Trees - Execution Time

- How to characterize the **execution time of each operation**?
  - All operations search for a node traversing the **height** of the tree

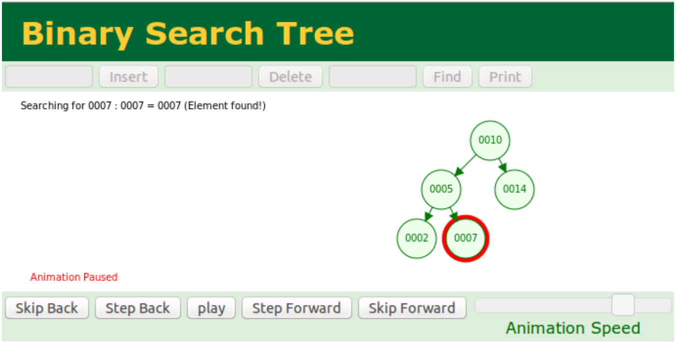
Complexity of operations in a binary search tree

Let h be the height of a binary search tree T. The complexity of finding the minimum, maximum, or searching for an element, or inserting or removing an element in T is $\mathcal{O}(h)$.

Binary Search Trees - Visualization

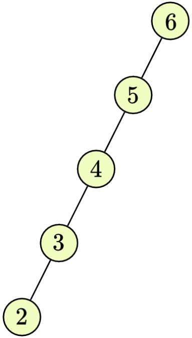
- A nice visualization of search, insertion and removal can be seen in:

  https://www.cs.usfca.edu/~galles/visualization/BST.html



Unbalanced Trees

- The **problem** of the previous methods:



The height of the tree can be of the order of $\mathcal{O}(n)$
(where n is the number of elements)

Balancing Strategies

- There are many strategies to guarantee that the complexity of the search, insertion and removal operations are better than $\mathcal{O}(n)$

Balanced Trees:
(height $\mathcal{O}(\log n)$)

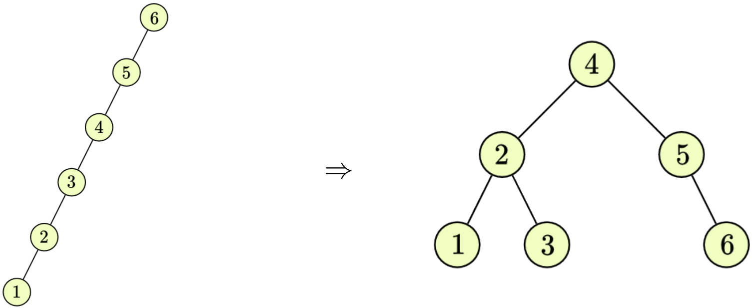
- AVL Trees
- Red-Black Trees
- Splay Trees
- Treaps

Other Data Structures:

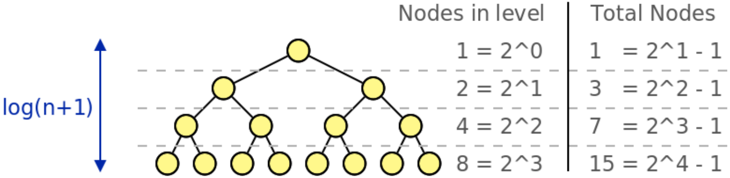
- Skip Lists
- Hash Tables
- Bloom Filters

Balancing Strategies

- A simple strategy: **reconstruct the tree** once in a while



- On a "perfect" binary tree with n nodes, the height is... $\mathcal{O}(\log(n))$



Balancing Strategies

Given a sorted list of numbers, **in which order should we insert them** in a binary search tree so that it stays as balanced as possible?

Answer: "binary search", insert the element in the middle, split the reamaining list in two (smaller and bigger) based on that element and insert each half applying the same method

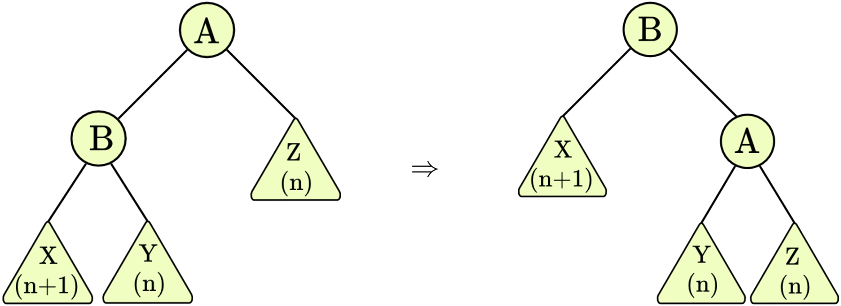
How frequently should we reconstruct the binary search tree so that we can guarantee efficiency?

- If we reconstruct often we have many $\mathcal{O}(n)$ operations
- If we rarely reconstruct, the tree may become unbalanced

A possible answer: after $\mathcal{O}(\sqrt{N})$ insertions

Balancing Strategies

- Simple case: **how to balance** the following tree (between parenthesis is the height):

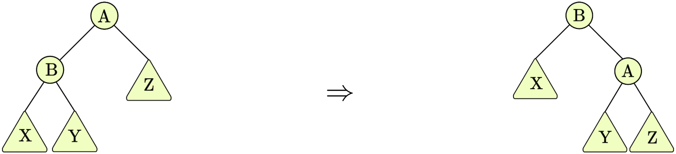


This operation is called a **right rotation**

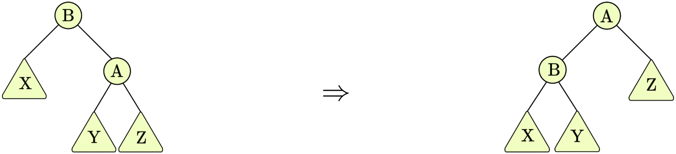
Balancing Strategies

- The relevant rotation operations are the following:
  - Note that we must not break the properties that turn the tree into a binary search tree

Right Rotation



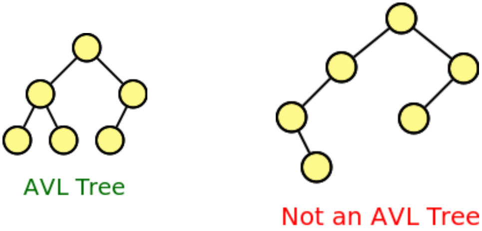
Left Rotation



AVL Trees

AVL Tree

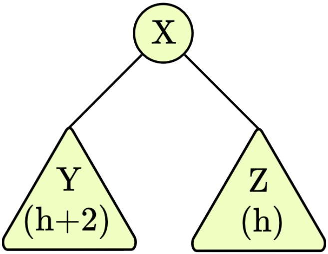
A binary search tree that guarantees that for each node, the heights of the left and right subtrees **differ by at most one unit** (**height invariant**)



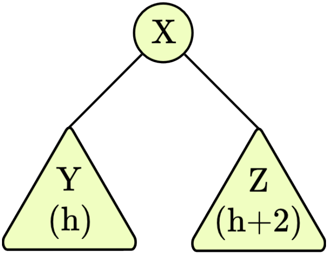
- When inserting and removing nodes, we change the tree so that we keep the **height invariant**

AVL Trees

- **Inserting** on a AVL tree works like inserting on any binary search tree. However, the tree might break the height invariant (and stop being "balanced")
- The following cases may occur:



+2 on the left

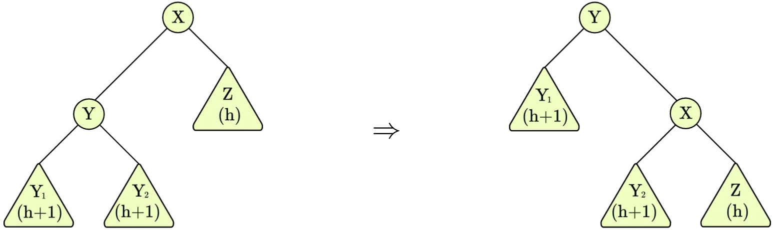


+2 on the right

- Let's see how to correct the first case with simple rotations. **Correcting the second case is similar**, but with mirrored rotations

AVL Trees

- In the first case, we have two different possible shapes of the AVL Tree
- The first:



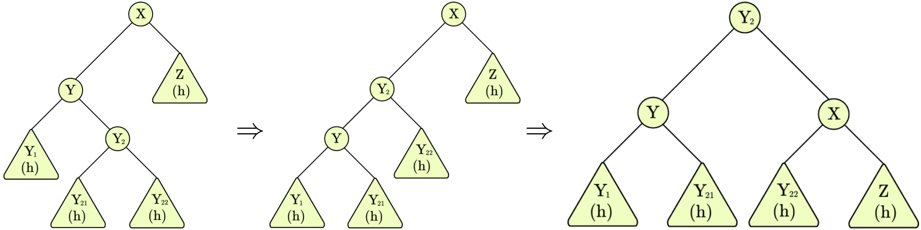
Left is too "heavy", case 1

We correct by making a right rotation starting in X

- Note: the height of Y_2 might be $h + 1$ or h: this correction works for both cases

AVL Trees

- The second:



Left is too "heavy", case 2

We correct by making a left rotation starting in Y, followed by a right rotation starting in X

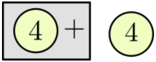
- Note: the height of Y_{21} **or** Y_{22} might be h or $h-1$: this correction works for both cases

AVL Trees

- By inserting nodes we might **unbalance** the tree (breaking the height invariant)
- In order to correct this, we apply rotations **along the path** where the node was inserted
- There are **two analogous unbalancing types**: to the left or to the right
- Each type has **two possible cases**, that are solved by applying different rotations

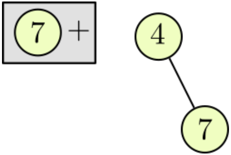
AVL Trees

- **Example** of node insertion:



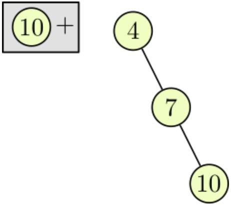
AVL Trees

- **Example** of node insertion:



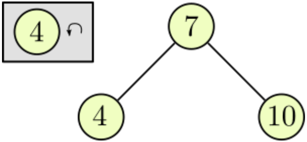
AVL Trees

- **Example** of node insertion:



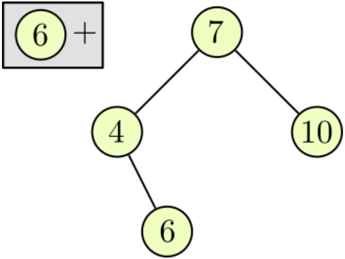
AVL Trees

- **Example** of node insertion:



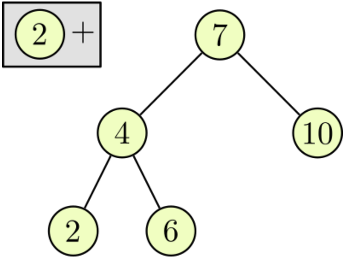
AVL Trees

- **Example** of node insertion:



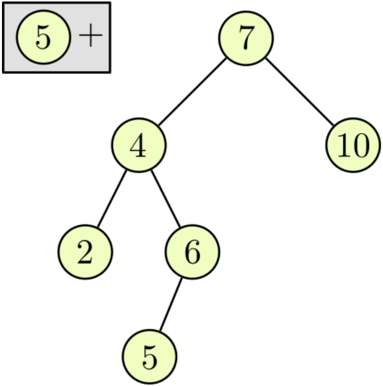
AVL Trees

- **Example** of node insertion:



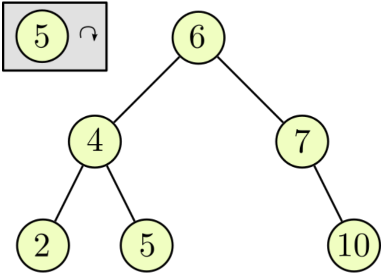
AVL Trees

- **Example** of node insertion:



AVL Trees

- **Example** of node insertion:



(after two rotations)

AVL Trees

- To **remove elements**, we apply the same idea of insertion
- First, we find the node to remove
- We apply one of the modifications seen for binary search trees
- We apply rotations as described along the path until we reach the root

AVL Trees

- For the **search** operation, we only traverse the tree height
- For the **insertion** operation, we traverse the tree height and the we apply at most two rotations (why only two?), that take $\mathcal{O}(1)$
- For the **removal** operation, we traverse the tree height and the we apply at most two rotations over the path until the root
- We conclude that the complexity of each operation is $\mathcal{O}(h)$, where h is the tree height

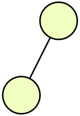
What is the maximum height of an AVL Tree?

AVL Trees

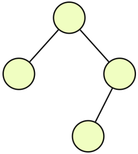
- To calculate the **worst case** of the tree height, let's do the following exercise:
  - What is the smallest AVL tree (following the height invariant) with height exactly h?
  - We will call $N(h)$ to the number of nodes of a tree with height h

AVL Trees

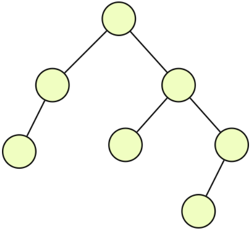
Height 1



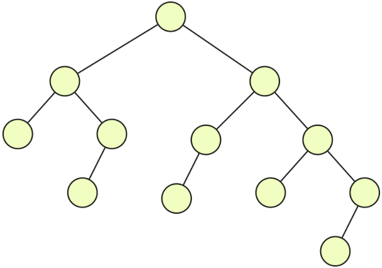
Height 2



Height 3



Height 4



Height 5

AVL Trees

- Summarizing:
  - $N(1) = 1$
  - $N(2) = 2$
  - $N(3) = 4$
  - $N(4) = 7$
  - $N(5) = 12$
  - $\dots$
  - $N(h) = N(h-2) + N(h-1) + 1$
- It has a behavior similar to the Fibonacci sequence!
- Remembering your linear algebra courses:
  - $N(h) \approx \phi^h$
  - $\log(N(h)) \approx \log(\phi)h$
  - $h \approx \frac{1}{\log(\phi)} \log(N(h))$

The height h of an AVL Tree with n nodes obeys to $h \leq 1.44 \log(n)$

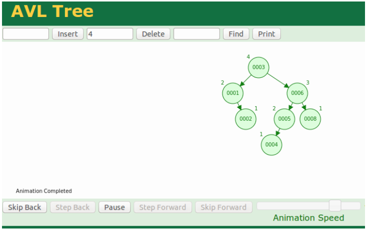
AVL Tree

- **Advantages** of AVL Trees:
  - Search, insertion and removal operations with guaranteed worst case complexity of $\mathcal{O}(\log n)$;
  - Very efficient search (when comparing with other related data structures), because the height limit of $1.44\log(n)$ is small;
- **Disadvantages** of AVL trees:
  - Complex implementation (we can simplify removal by using *lazy delete*, similar to the idea of reconstructing);
  - Implementation requires two extra *bits* of memory per node (to store the "unbalancedness" of a node: $+1$, 0 or -1);
  - Insertion and removal less efficient (when comparing with other related data structures) because of having to guarantee a smaller maximum height;
  - The rotations frequently change the tree structure (not cache or disk friendly);

AVL Trees

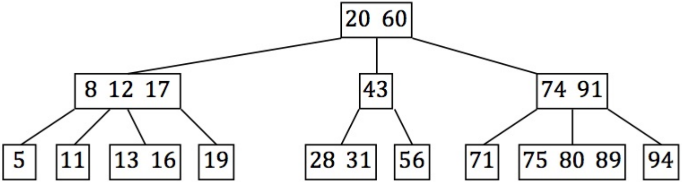
- The name AVL comes from the authors: G. **A**delson-**V**elsky and E. **L**andis. The original paper describing them is from 1962 (*"An algorithm for the organization of information"*, Proceedings of the USSR Academy of Sciences)

- You can use an AVL Tree visualization to "play" a little bit with the concept and seeing how are insertions, removals and rotations made. `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`



Red-Black Trees

- We will now explore another type of binary search trees known as **red**-**black** trees
- This type of trees appeared as an "adaptation" of **2-3-4 trees** to binary trees



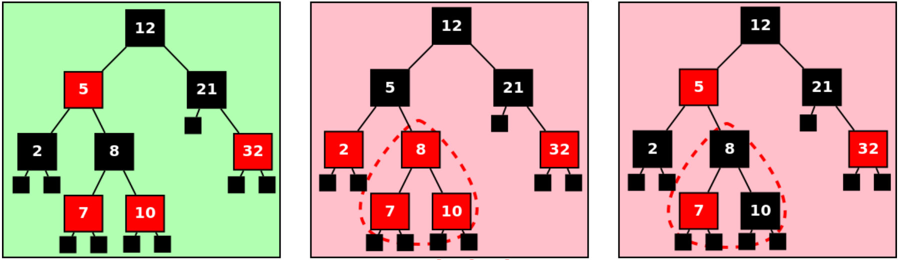
- The original paper is from 1978 and was it was written by L. Guibas e R. Sedgewick ("*A Dichromatic Framework for Balanced Trees*")
- The authors say they use the red and black colors because they looked good when printed and because those were the pen colors thay they had available to draw the trees :)

Red-Black Trees

Red-Black Tree

A binary search tree where each node is either black or red and:

- **(root property)** The root node is black
- **(leaf property)** The leaves are null/empty black nodes
- **(red property)** The children of a red node are black
- **(black property)** For each node, a path to any of its descending leaves has the same number of black nodes



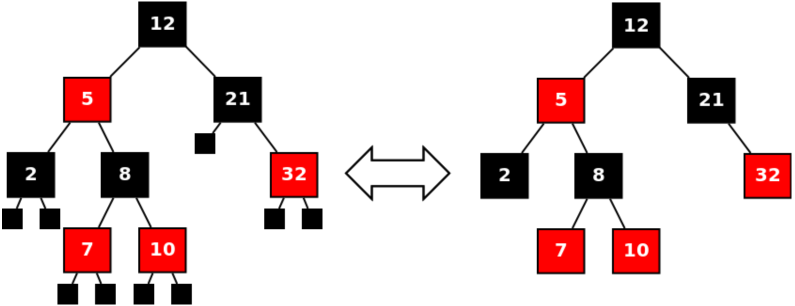
Red-Black Tree

Not a Red-Black Tree
(missing "red property")

Not a Red-Black Tree
(missing "black property")

Red-Black Trees

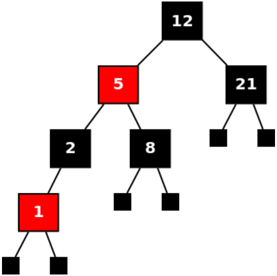
- For better visibility, the images may not contain the "null" leaves, but you may assume those nodes exist.
  We call **internal nodes** to the non null nodes.



- The number of black nodes in a path from a node n to any of its leaves (not including the node itself) is known as **black height** and will be denoted as $bh(n)$
  - Ex: $\rightarrow bh(12) = 2$ and $bh(21) = 1$

Red-Black Trees

- What type of balance do the restrictions guarantee?
- If $bh(n) = k$, then a path from n to a leaf has:
  - At least k nodes (only black nodes)
  - At most $2k$ nodes (alternating between black and red nodes) *[recall that there are never two consecutive red nodes]*
- The height of a branch is therefore at most double the height of a sister branch



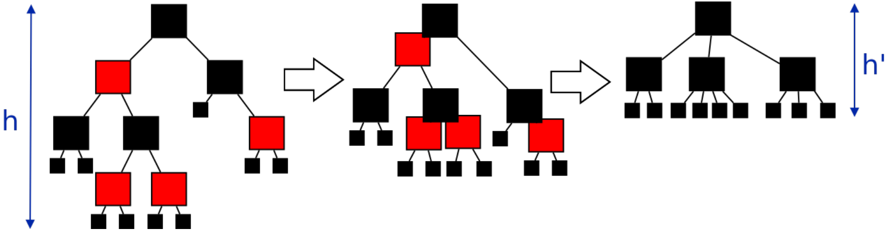
Red-Black Trees

Theorem - Height of a Red-Black Tree

A red-black tree with n nodes has height $h \leq 2 \times \log_2(n+1)$
[that is, the height h of a red-black tree is $\mathcal{O}(\log n)$*]*

Intuition:

Let's *merge* the red nodes with their black parents:

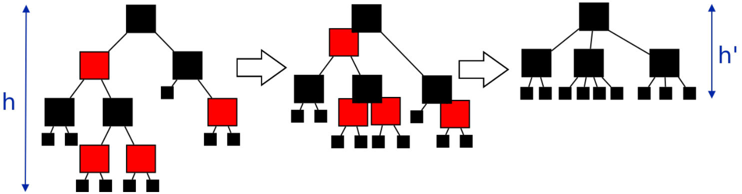


- This process produces a tree with 2, 3 or 4 children
- This 2-3-4 tree has leaves at an uniform height of **h'** (where h' is the *black height*)

Red-Black Trees

Theorem - Height of a Red-Black Tree

A red-black tree with n nodes has height $h \leq 2 \times \log_2(n+1)$
[that is, the height h of a red-black tree is $\mathcal{O}(\log n)$]



- The height of this tree is at least half of the original: $h' \geq h/2$
- A complete binary tree of height h' has $2^{h'} - 1$ internal (non null) nodes
- The number of internal nodes of the new tree is $\geq 2^{h'} - 1$ (it is a 2-3-4 tree)
- The original tree had even more nodes than the new one: $n \geq 2^{h'} - 1$
- $n + 1 \geq 2^{h'}$
- $\log_2(n+1) \geq h' \geq h/2$
- $h \leq 2\log_2(n+1)$ $\square$

Red-Black Trees

- How to make an **insertion**?

Inserting a node in a non empty red-black tree

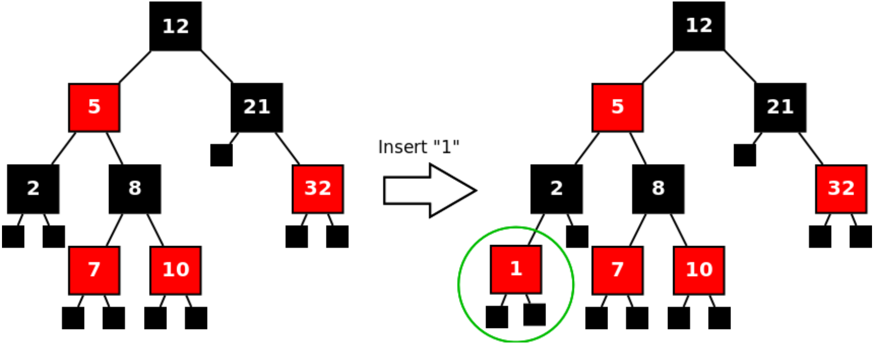
- Insert as in any binary search tree
- Color the inserted node as red (adding the null black nodes)
- Recolor and restructure if needed (restore the invariants)

- Because the tree is non empty we don't break the **root property**
- Because the inserted node is red, we don't break the **black property**
- The only invariant than can be broken is the **red property**
  - If the parent of the inserted node is **black**, nothing needs to be done
  - If the parent is **red** we now have two consecutive red nodes

Red-Black Trees

When the parent of the inserted node is **black** nothing needs to be done:

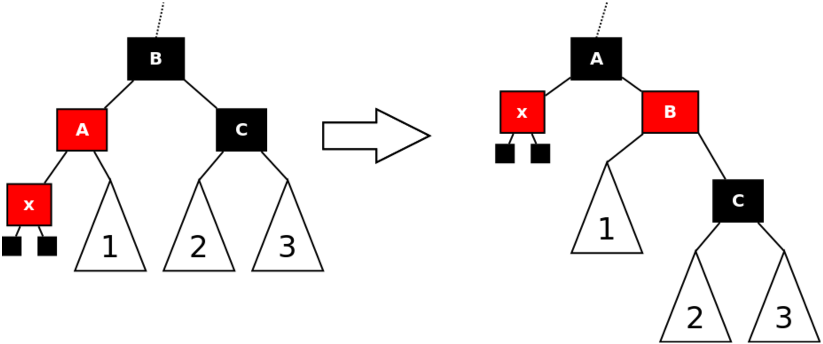
Example:



Red-Black Trees

Red-Red after insertion (red parent)

- Case 1.a) The uncle is a **black** node and the inserted node x is the left child

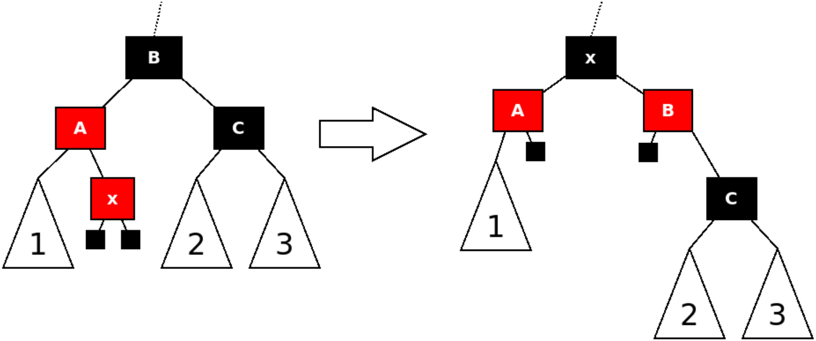


Description: right rotate the grandfather, followed by swapping the colors between the parent and the grandfather

Red-Black Trees

Red-Red after insertion (red parent)

- Case 1.b) The uncle is a **black** node and the inserted node x is the right child



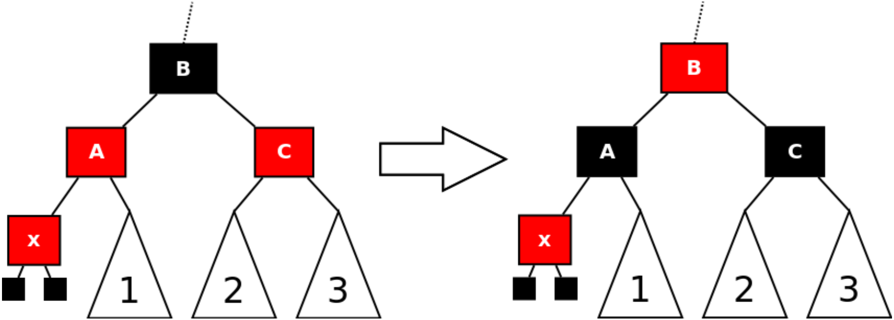
Description: left rotation of parent followed by the moves of 1.a

[If the parent was the right child of the grandfather, we would have similar cases, but symmetric in relation to these]

Red-Black Trees

Red-Red after insertion (red parent)

- Case 2: The uncle is a **red** node, with x being the inserted node



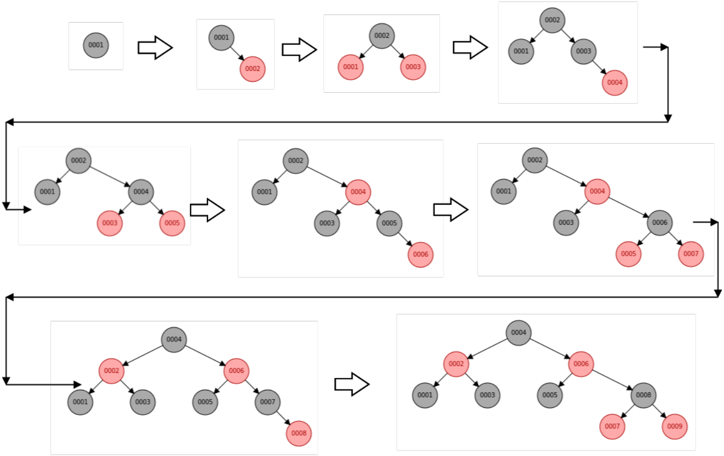
Description: swap colors of parent, uncle and grandfather

Now, if the father of the grandfather is red, we have a new red-red situation and we can simply apply one of the cases we already know (if the gradparent is the root, we simply color it as black)

Red-Black Trees

- Let's visualize some insertions (try the indicated url):

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html



Red-Black Trees

- The cost of an **insertion** is therefore $\mathcal{O}(\log n)$
  - $\mathcal{O}(\log n)$ to get to the insertion position
  - $\mathcal{O}(1)$ to eventually recolor and restructure

- The **removals** are similar albeit a bit more complicated, but they also cost $\mathcal{O}(\log n)$
  (we will not detail in class, but you can try the visualizations)

Red-Black Trees

- **Comparison** of Red-Black Trees (RB) with AVL trees
  - Both are implemented with balanced binary search trees (search, insertion and removal are $\mathcal{O}(\log n)$)
  - RB are a little bit more unbalanced in the worst case, with height $\sim 2\log(n)$ *vs* AVL with height $\sim 1.44\log(n)$
  - RB may take a little bit more time to search (at the worst case, because of the height)
  - RB are a bit faster in insertions/removals on average ("lighter" rebalancing)
  - RB spend less memory (RB only need 1 extra bit for color, AVL 2 bits for unbalancedness)
  - RB are (probably) more used in the classical programming languages Examples of data structures that use them:
    - ★ C++ STL: set, multiset, map, multiset
    - ★ Java: java.util.TreeMap , java.util.TreeSet
    - ★ Linux kernel: scheduler, linux/rbtree.h

Use in C/C++, Java and other languages

- Any typical programming language has an implementation of balanced binary search trees *(with guaranteed logarithmic time)*
- The associated main data structures are:
  - **set**: search, insert and remove elements
  - **multiset**: a *set* with possibly repeated elements
  - **map**: associative array (associates a key with a value) ex: associating *strings* to *ints*)
  - **multimap**: a *map* with the possibility of repeated keys
- The nodes may contain any data types as long as they are **comparable**
- Because there is relative order between nodes, you can use **iterators** to traverse the trees in order (ex: in increasing order, from min to max)

Tree Data Structures

- Besides **AVL** and **Red-Black trees**, there are many other types of binary search trees that have different characteristics.
- More than that, **tree data structures are ubiquitous in Computer Science** and they are used for many purposes, being a very powerful and flexible topology.

V • T • E	Tree data structures [hide]
Search trees (dynamic sets/associative arrays)	2–3 • 2–3–4 • AA • (a,b) • AVL • B • B+ • B* • B^x • (Optimal) Binary search • Dancing • HTree • Interval • Order statistic • (Left-leaning) Red–black • Scapegoat • Splay • T • Treap • UB • Weight-balanced
Heaps	Binary • Binomial • Brodal • Fibonacci • Leftist • Pairing • Skew • van Emde Boas • Weak
Tries	Ctrie • C-trie (compressed ADT) • Hash • Radix • Suffix • Ternary search • X-fast • Y-fast
Spatial data partitioning trees	Ball • BK • BSP • Cartesian • Hilbert R • *k*-d (implicit *k*-d) • M • Metric • MVP • Octree • Priority R • Quad • R • R+ • R* • Segment • VP • X
Other trees	Cover • Exponential • Fenwick • Finger • Fractal tree index • Fusion • Hash calendar • iDistance • K-ary • Left-child right-sibling • Link/cut • Log-structured merge • Merkle • PQ • Range • SPQR • Top

`https://en.wikipedia.org/wiki/Tree_(data_structure)`

- In this course we will explore other trees (and their different conceptual approaches):
  - Splay Trees (and amortized analysis)
  - Treaps (and probabilistic analysis)
  - Quadtrees, kd-trees and 2D range trees (as spatial trees)
- We will now have a quick look at one other type of search tree, to present its key ideas and usages

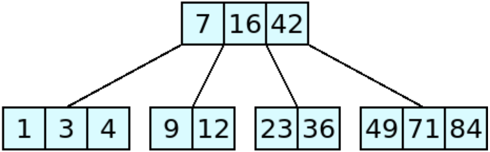
B-Trees

- A **self-balancing search tree** that can have more than 2 children per node
- **Motivation**: **minimize number of disk accesses if data is stored on disk**
- **Key idea**: nodes with many elements so that they may correspond to a disk page (minimizing tree traversal between nodes)
- Introduced by **R. Bayer** and and **E. McCreight** in **1970**
("Organization and maintenance of large ordered indexes")
- Provide guarantees of **logarithmic** operations

- Sometimes the term is used to refer to a class of balanced tree data structures: B-Tree, B+Tree, B*Tree, B^{link}-tree
- Terminology may vary, but here we will use the term to refer to a specific data structure

B-Trees - A possible definition

- A **B-Tree** of order m satisfies the following **properties**:
  - Every node has at most m children.
  - Every non-leaf node (except the root) has at least $\frac{m}{2}$ child nodes
  - A non-leaf node with k children contains $k - 1$ keys.
  - All leaves appear in the same level (they have the the same depth) (the tree is always "perfectly balanced")



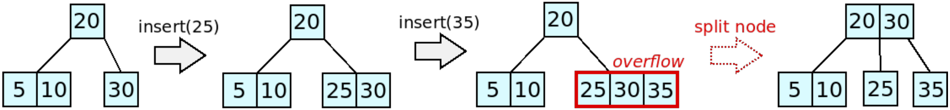
An example B-Tree of order 4

(some literature would say the order is 2, as in a b-tree of order d can have at most $2d$ children)

Operations on a B-Tree

- **find(x):** standard BST-type walk down the tree
- **insert(x):** insert in a leaf as in a BST, increasing the number of keys in the node; if the node *overflows*, split in two and the middle element is inserted to parent (a cascade of splits may occur)
- **remove(x):** find the node and remove that key; if the node *underflows*, it may borrow some elements from neighboring nodes or, if the nodes are small, they may be merged
  (this is a very simplified explanation)

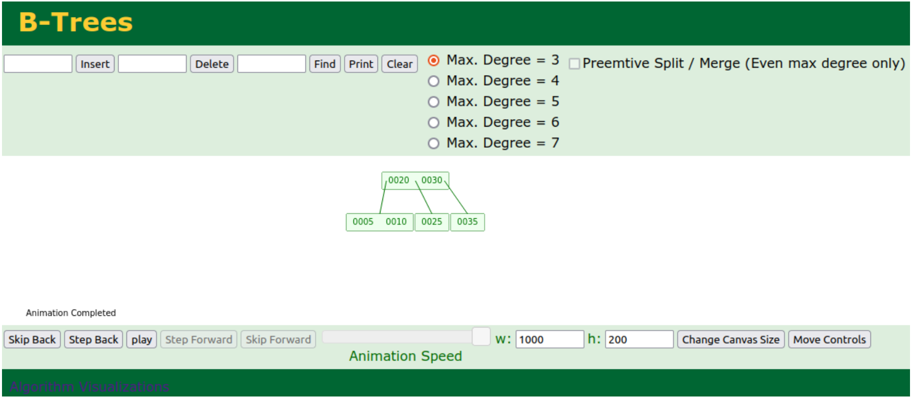
Example insertions in a B-Tree of order 3*:*



Visualizing B-Trees

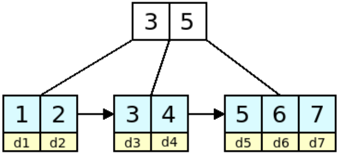
- You can try the indicated url:

  https://www.cs.usfca.edu/~galles/visualization/BTree.html



B+Trees - A possible definition

- A **B+Tree** is a variant of a B-Tree in which:
  - Data is only stored on leafs (internal nodes only have keys)
  - The leaves have links to their siblings



An example B+Tree: in the leaves each key i has associated data d_i

(think of pairs (key,data) as in STL maps)

- The lower (leaf) level allows for quick traversal of ranges

B-Trees in real life

- Specialized B-Trees and their variants are still used for indexing in many real-life systems:
  - In **filesystems** such as Windows NTFS, Linux ext3 or MacOS APFS
  - In **relational Databases** such as MySQL, MariaDB or PostgreSQL
- Typically use **large block sizes** (order of the b-tree), matching real disk blocks and leading to a really small tree height