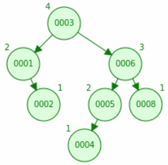
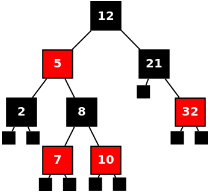
Balanced Binary Search Trees

L.EIC

Algorithms and Data Structures

2025/2026

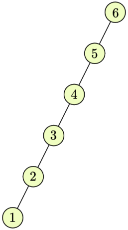




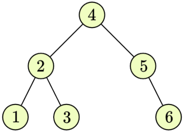
P Ribeiro, AP Tomas

Binary Search Tree (BST) - A quick recap

- **Every** node in a BST: **greater than all the nodes in its left subtree and smaller than all the nodes in its right subtree**
- The **time complexity** of *naively* inserting, removing and searching for elements in a BST is $\mathcal{O}(h)$, where h is the height of the tree
- The height depends on the insertion order and a **bad order** may give origin to a height that is **linear on the number of elements** n
- However, if the tree is **balanced**, the height is $\mathcal{O}(\log n)$



vs



Balancing Strategies

- There are several strategies to ensure that the complexity of operations like searching, inserting, and removing is better than $\mathcal{O}(n)$

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

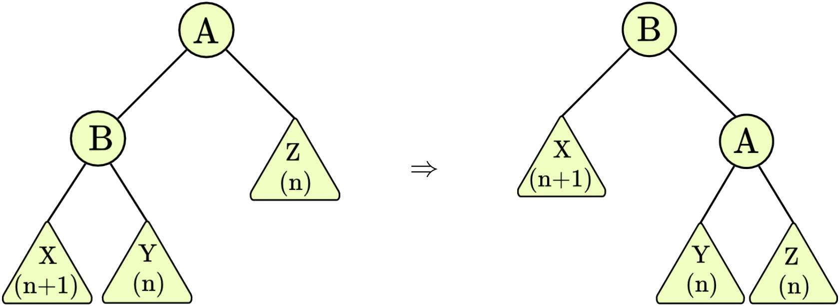
- AVL Trees *(in detail today)*
- Red-Black Trees *(in detail today)*
- Splay Trees *(quick overview today)*
- B-Trees *(quick overview today)*
- Treaps
- ...

Other data structures:

- Skip List
- Hash Table *(on another class)*
- Bloom Filter

Balancing Strategies

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

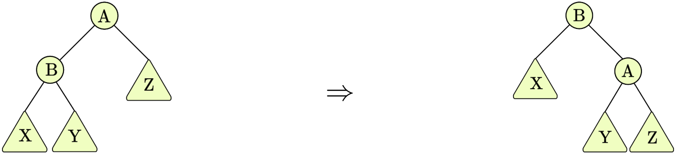


This operation is called a **right rotation**

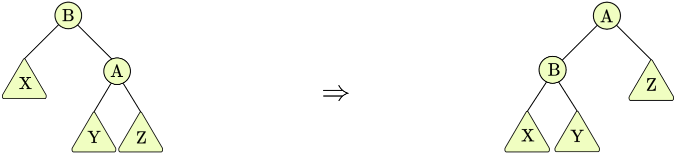
Balancing Strategies

- 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



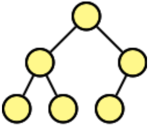
Left Rotation



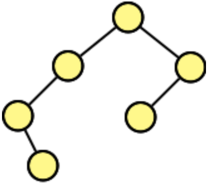
AVL Trees

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**)



AVL Tree

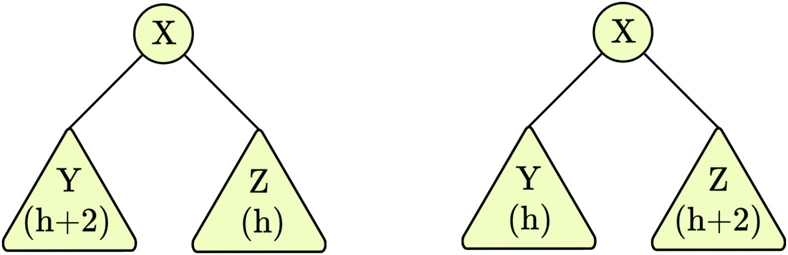


Not an AVL Tree

- 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:



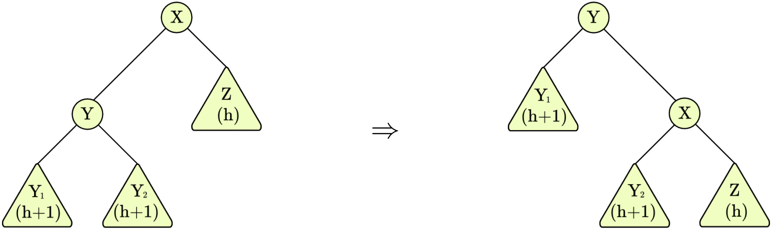
+2 on the left

+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:



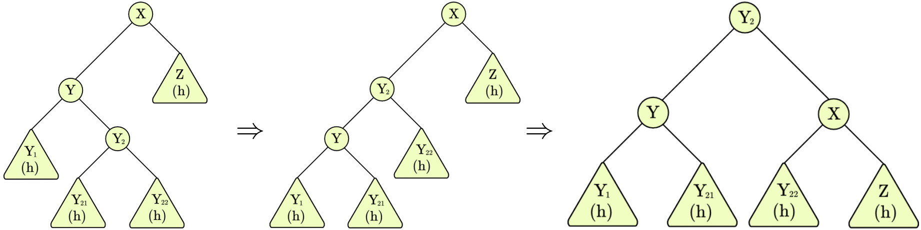
Left is too "heavy", case 1

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

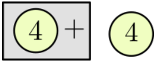
We correct by making a left rotation starting in Y, followed by a right rotation starting in X

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

AVL Trees

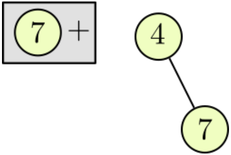
- By inserting nodes we might **unbalance** the tree (breaking the height invariant)
- In order to correct this, we apply rotations **along the path** where the node was inserted
- There are **two analogous unbalancing types**: to the left or to the right
- Each type has **two possible cases**, that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

- **Example** of node insertion:



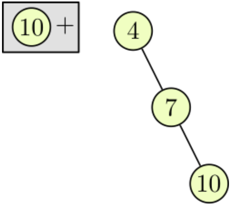
AVL Trees

- **Example** of node insertion:

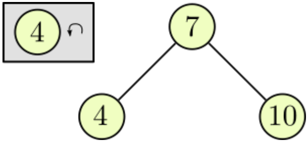


AVL Trees

- **Example** of node insertion:

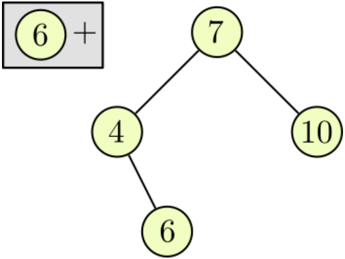


- **Example** of node insertion:



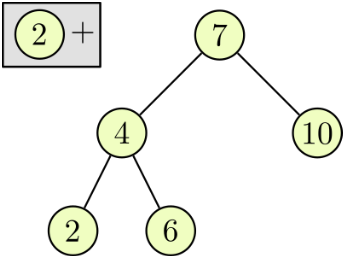
AVL Trees

- **Example** of node insertion:



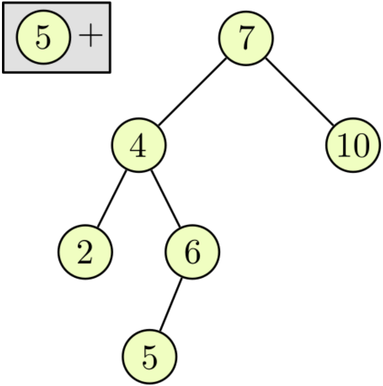
AVL Trees

- **Example** of node insertion:



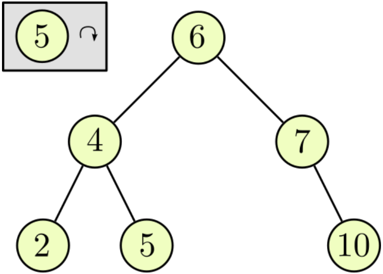
AVL Trees

- **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 the same process we have seen for binary search trees
- We apply rotations as described along the path on all unbalanced nodes 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 then we apply $\mathcal{O}(h)$ 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

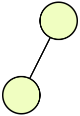
What is the maximum height of an AVL Tree?

AVL Trees

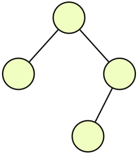
- To calculate the **worst case** of the tree height, let's do the following exercise:
  - What is the smallest AVL tree (following the height invariant) with height exactly h?
  - We will call $N(h)$ to the number of nodes of a tree with height h

AVL Trees

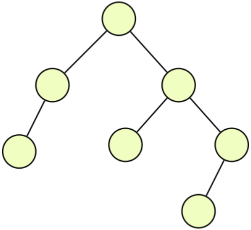
Height 1



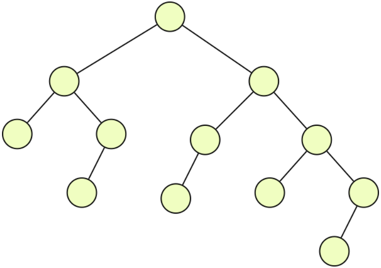
Height 2



Height 3



Height 4



Height 5

AVL Trees

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

The height h of an AVL Tree with n nodes is roughly (at most) $1.44 \log(n)$, which is $\mathcal{O}(\log n)$

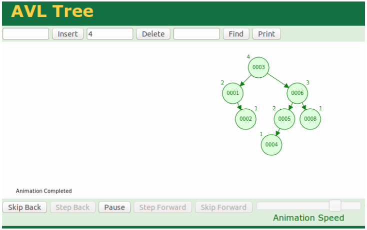
AVL Tree

- **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:
  - Relatively complex implementation (still simpler than other similar data structures);
  - 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

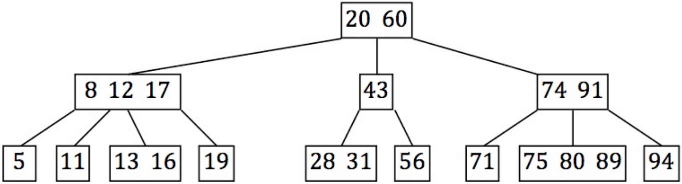
- The name AVL comes from the authors: G. **A**delson-**V**elsky and E. **L**andis. The original paper describing them is from 1962 (*"An algorithm for the organization of information*", Proceedings of the USSR Academy of Sciences)

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



Red-Black Trees

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



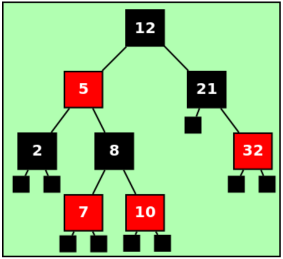
- The original paper is from 1978 and was it was written by L. Guibas e R. Sedgewick (*"A Dichromatic Framework for Balanced Trees"*)
- The authors say they use the red and black colors because they looked good when printed and because those were the pen colors thay they had available to draw the trees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

Red-Black Trees

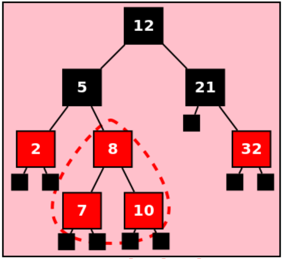
Red-Black Tree

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

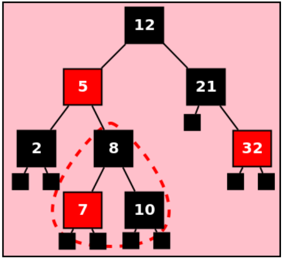
- **(root property)** The root node is black
- **(leaf property)** The leaves are null/empty black nodes
- **(red property)** The children of a red node are black
- **(black property)** For each node, a path to any of its descending leaves has the same number of black nodes



Red-Black Tree



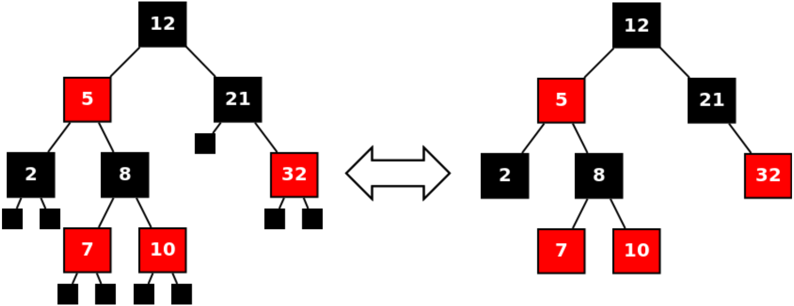
Not a Red-Black Tree
(missing "red property")



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

Red-Black Trees

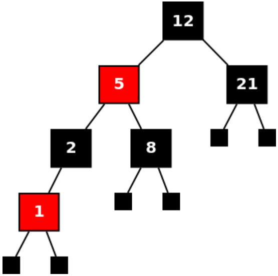
- For better visibility, the images may not contain the "null" leaves, but you may assume those nodes exist.
  We call **internal nodes** to the non null nodes.



- 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



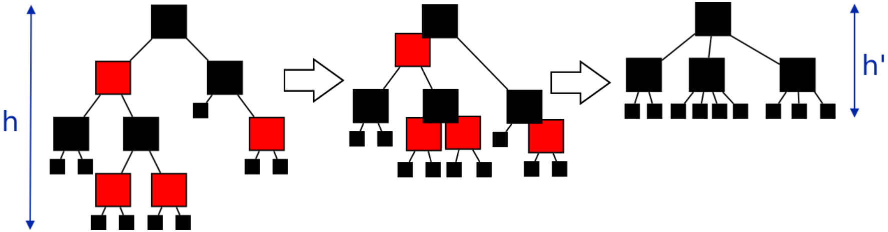
Red-Black Trees

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:

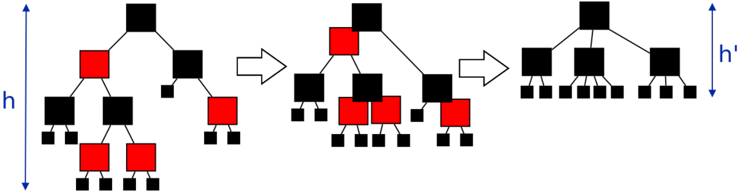


- This process produces a tree with 2, 3 or 4 children
- This 2-3-4 tree has leaves at an uniform height of **h'** (where h' is the *black height*)

Red-Black Trees

Theorem - Height of a Red-Black Tree

A red-black tree with n nodes has height $\mathrm{h} \leq 2 \times \log_2(\mathrm{n}+1)$
[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: $n \geq 2^{h'} - 1$
- $n + 1 \geq 2^{h'}$
- $\log_2(n+1) \geq h' \geq h/2$
- $h \leq 2\log_2(n+1)$ □

Red-Black Trees - A quick recap

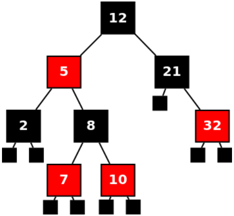
Red-Black Tree

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

Theorem - Height of a Red-Black Tree

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



Intuition:

- the black property and the black nodes guarantee "balance" (black height is equal in all nodes)
- the red nodes are the allowed "lack of balance" (and no two consecutive red nodes are allowed)

Red-Black Trees

- How to make an **insertion**?

Inserting a node in a non empty red-black tree

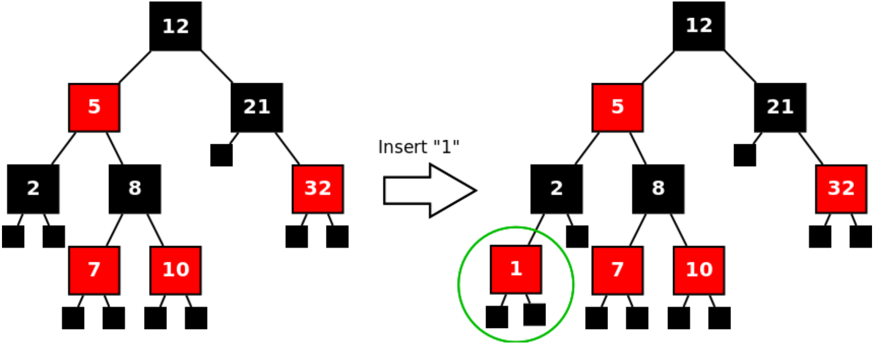
- Insert as in any binary search tree
- Color the inserted node as red (adding the null black nodes)
- Recolor and restructure if needed (restore the invariants)

- 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:

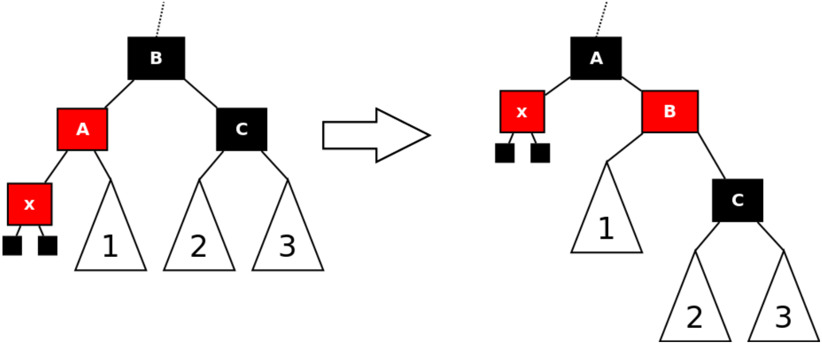
Example:



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

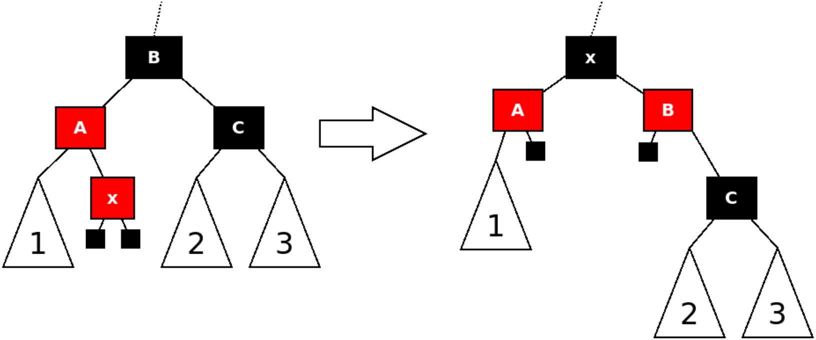


Description: right rotate the grandfather, followed by swapping the colors between the parent and the grandfather

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



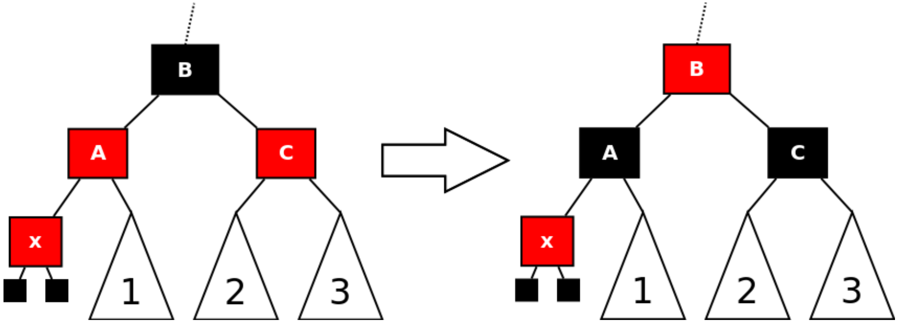
Description: left rotation of parent followed by the moves of 1.a

[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 <u>uncle</u> is a **red** node, with x being the inserted node



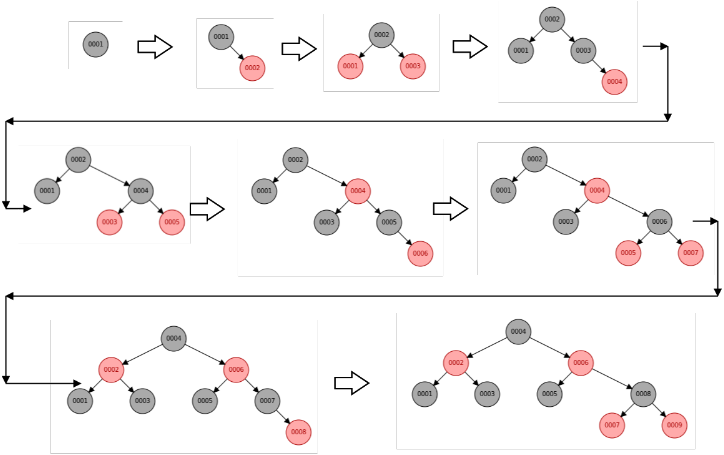
Description: swap colors of parent, uncle and grandfather

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 grandparent 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, multimap
    - Java: java.util.TreeMap , java.util.TreeSet
    - Linux kernel: scheduler, linux/rbtree.h

A note about C++

- **Red-black trees are used nowadays in most common C++ compilers** but that does not mean they will always be used
- The standard only "demands" $\mathcal{O}(\log n)$ for the common set and map operations and "BST like" iterators
- It is impossible to be "perfect" for all situations (e.g. should we expect more insertions, deletions or searches?)

- Languages are **dynamic and always evolving**; C++ is no exception
- Languages gain new constructs, libraries, requirements, etc.
- Last Standards: C++23, C++20, C++17, C++14,
- C++26 will be the next version
- The C++ Standards Committee / Boost C++ Libraries

Using (already implemented) BSTs in C++

- **(Ordered) Associative Containers**
  - `set` - collection of unique keys, sorted by keys
  - `map` - collection of key-value pairs, sorted by keys, keys are unique
  - `multiset` - collection of keys, sorted by keys
  - `multimap` - collection of key-value pairs, sorted by keys
- Usual operations are available:
  - Iterators (forward and reverse)
  - Lookup (find, count, lower_bound, upper_bound, ...)
  - Modifiers (clear, insert, erase, ...)

Example Applications

- Let's do some **livecoding** and use a **real dataset** to play a little bit
- Imagine you have all students first names on a file `names.txt` *(possibly with repetitions)*

```
Fernando
Jose
Marcos
Vasco
...
```

```cpp
// Example that reads all strings from stdin and prints them (one per line)
string name;
while (cin >> name) {
  cout << name << endl;
}
```

Example compilation using gcc:

```
g++ -o example example.cpp
```

Example execution (< redirects stdin, `./` indicates current dir)

```
./example < names.txt
```

Example Applications

- Calculating how many different names exist?

```
set<string> s;    // Set to contain all different names
string name;
while (cin >> name) {
  s.insert(name); // Insert all names on the set (keys should be unique)
}
cout << "There are " << s.size() << " different names" << endl;
```

```
There are 132 different names
```

Time complexity: $\mathcal{O}(n \log n)$

- Notice the difference when using a multiset (also in time $\mathcal{O}(n \log n)$):

```
multiset<string> ms; // now we are using a multiset
string name;
while (cin >> name) {
  ms.insert(name);
}
cout << "There are " << ms.size() << " names" << endl;
```

```
There are 400 names
```

Example Applications

Here are some more examples of available methods of the container `set`:

- Searching for an element (in time $\mathcal{O}(\log n)$) [C++20 introduces `contains()`]:

```
string n1 = "Pedro";
if (s.find(n1)!=s.end()) cout << n1 << " found" << endl;
else cout << n1 << " not found" << endl;
string n2 = "Aniceto";
if (s.find(n2)!=s.end()) cout << n2 << " found" << endl;
else cout << n2 << " not found" << endl;
```

```
Pedro found
Aniceto not found
```

- Traversing the elements of the set, in increasing order (in time $\mathcal{O}(n)$):

```
// Range-based for loop (auto:  automatically deduce type)
for (auto i : s) {
  cout << i << endl;
}
```

```
Abecassis
Adriana
Afonso
...
```

Example Applications

Here are some more examples of available methods of the container `set`:

- Using iterators (in time $\mathcal{O}(1)$ for each `begin()`, increment and decrement):

```
auto i = s.begin(); // Iterator starting with smallest element
cout << "1st name: " << *i << endl;
i++; cout << "2nd name: " << *i << endl;
i++; cout << "3rd name: " << *i << endl;
i--; cout << "2nd name: " << *i << endl;
auto j = s.end(); // Start at the end [could have instead used rbegin()]
j--; cout << "Last name: " << *j << endl;
j--; cout << "Second to last name: " << *j << endl;
```

```
1st name: Abecassis
2nd name: Adriana
3rd name: Afonso
2nd name: Adriana
Last name: Zoe
Second to last name: Ye
```

What happens if you try to increment more than one unit?
(e.g. `i+=2`)

Example Applications

Here are some more examples of available methods of the container `set`:

- Erasing elements in various fashions:

```
cout << "size = " << s.size() << endl;
s.erase("Aniceto"); // Element does not exist, nothing is erased
cout << "size = " << s.size() << endl;
s.erase("Pedro");   // We can erase by key - time:  O(log n)
cout << "size = " << s.size() << endl;
s.erase(s.begin()); // We can erase by iterator - time:  O(1)
cout << "size = " << s.size() << endl;
// Below we take O(log n + k), where k is the number of elements to remove
s.erase(s.find("Carlos"), s.find("Sofia")); // We can erase a range
cout << "size = " << s.size() << endl;
```

```
size = 132
size = 132
size = 131
size = 130
size = 32
```

- There are many more methods: always look into to the documentation to check what exists and the how methods work

Example Applications

- What if we want the frequency of each name?

```
map<string, int> m; // maps a name to its frequency
string name;
while (cin >> name) {
  if (m.find(name)==m.end()) m[name] = 1; // new name
  else m[name]++; // existing name, just increment its frequency
}

// i becomes a pair (key, value), elements are sorted by key
for (auto i : m) {
  cout << i.first << " " << i.second << endl;
}
```

```
Abecassis 1
Adriana 1
Afonso 10
...
```

- How could you find the most frequent name?
  Can you guess what it is at this course?

Example Applications

- Like with sort, you can use a custom comparator:

```
// Example of using lambda functions (available since C++11)
auto comp_length = [](const string& a, const string& b) {
  return a.length() < b.length();
};
set<string, decltype(comp_length)> s(comp_length);
string name;
while (cin >> name) s.insert(name);
for (auto i : s) cout << i << endl;
```

```
Ye
Ana
Agda
Allan
Afonso
...
```

Example Applications

- And we can use our own custom classes and overload the < operator

```
class Person {
public:
  string name, surname;
  Person(string n, string s) {name=n; surname=s;}
};

bool operator< (const Person & p1, const Person & p2) {
  return p1.surname < p2.surname;
}
```

```
set<Person> s;
s.insert(Person("Ana","Tomas"));
s.insert(Person("Pedro","Ribeiro"));
s.insert(Person("Vasco","Cruz"));
s.insert(Person("Vanessa","Silva"));
for (auto i : s) cout << i.name << " " << i.surname << endl;
```

```
Vasco Cruz
Pedro Ribeiro
Vanessa Silva
Ana Tomas
```

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)

- The next slides provide a quick look at two other search trees, to present their key ideas and usages
  (you do not need to know splay trees and b-trees for AED evaluations)

Splay Trees

- A **self-adjusting binary search tree** that restructures the tree even when simply searching for an element
- **Motivation**: **provide quick access to recently accessed elements**
- **Key idea**: accessed items are moved to the root
- Introduced by **D. Sleator** and **R. Tarjan** in **1985** *("Self-Adjusting Binary Search Trees")*
- Provide guarantees of **logarithmic** operations in **amortized sense**

Amortized complexity

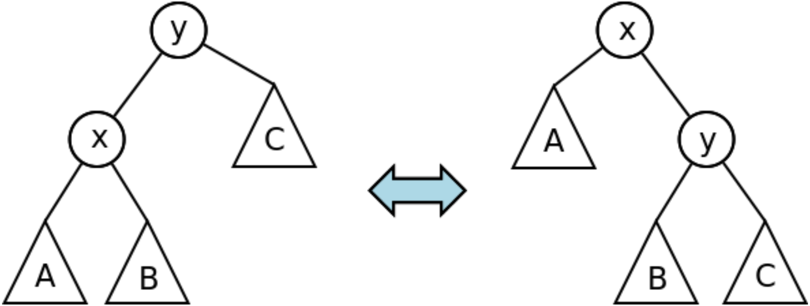
The amortized sequence complexity is the **worst case sequence complexity** (that is, the maximum possible total cost over all possible sequences of n operations) divided by n

(some operations may cost more, but others will cost less: on average they are $\mathcal{O}(\log n)$)

Splay Tree Rotations

- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

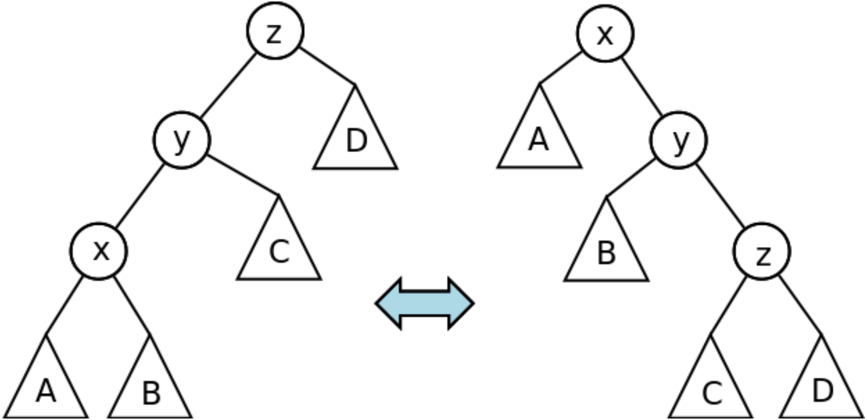
Zig (or **Zag**) - Simple Rotation
(also used in AVL and red-black trees)



Splay Tree Rotations

- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

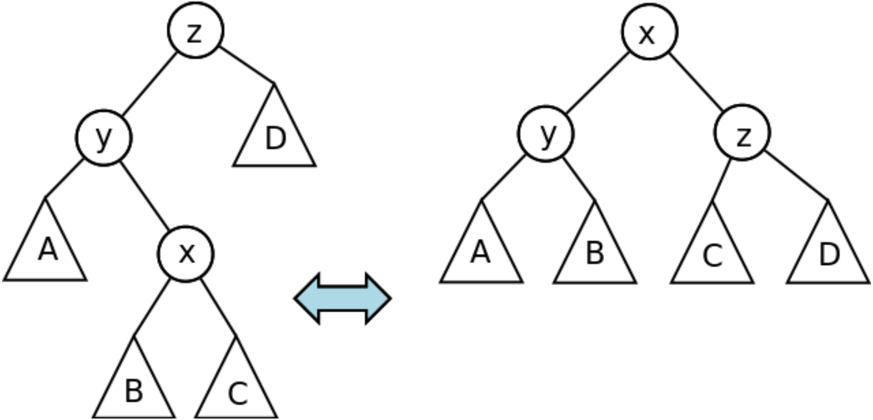
Zig-Zig (or **Zag-Zag**)



Splay Tree Rotations

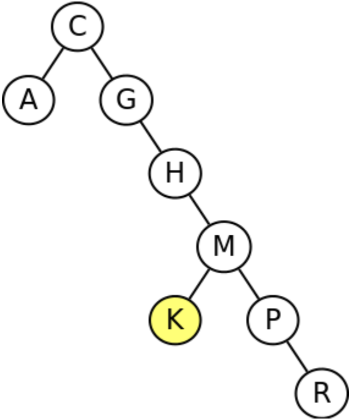
- Consider the following "rotations" designed to move a node to the root of a (sub)tree:

Zig-Zag (or **Zag-Zig**)



Splay Operation

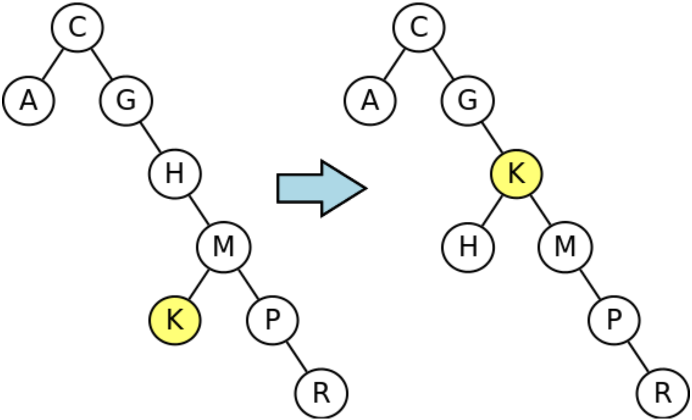
- Splaying a node means moving it to the root of a tree using the operations given before:



Original tree

Splay Operation

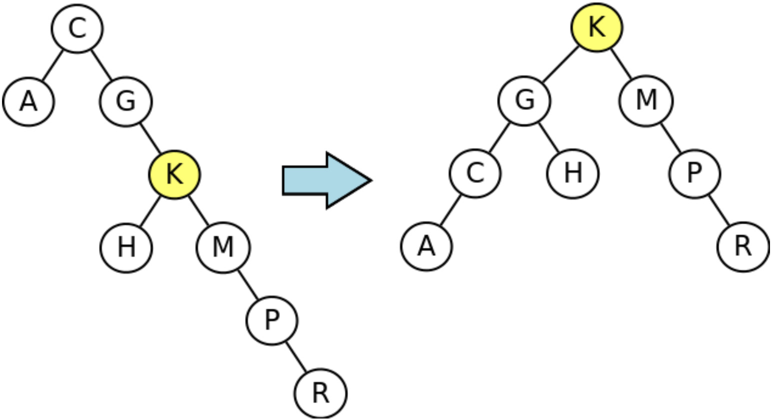
- Splaying a node means moving it to the root of a tree using the operations given before:



Zig-Zag Left (or Zag-Zig)

Splay Operation

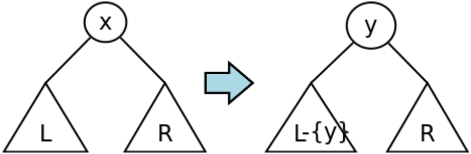
- Splaying a node means moving it to the root of a tree using the operations given before:



Zig-Zig Left (or Zag-Zag)

Operations on a Splay Tree

- **Idea:** do as in a normal BST but in the end splay the node
  - **find(x):** do as in BST and then splay x
    (if x is not present splay the last node accessed)
  - **insert(x):** do as in BST and then splay x
  - **remove(x):** find x, splay x, delete x (leaves its subtress R and L "detached"), find largest element y in L and make it the new root:



- Running time is **dominated** by the splay operation.

Why do Splay Trees work in practice?

Efficiency of splay trees

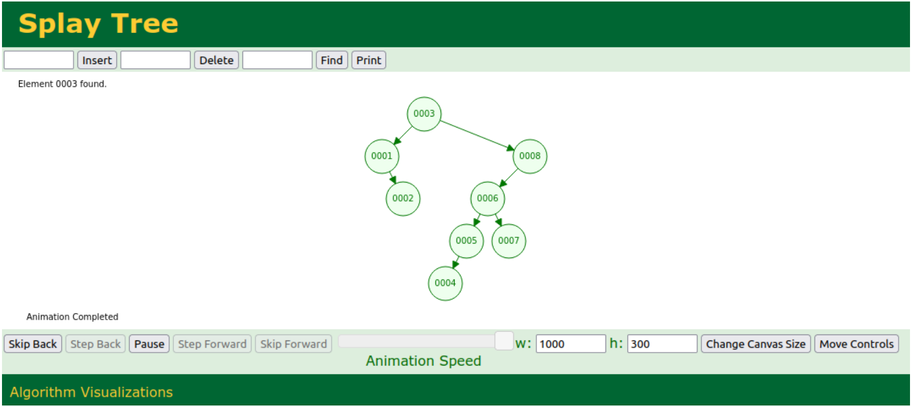
For any sequence of m operations on a splay tree, the running time is $\mathcal{O}(m \log n)$, where n is the max number of nodes in the tree at any time.

- **Intuition:** any operation on a deeper side of the tree will "bring" nodes from that side closer to the root
  - It is possible to make a splay tree have $\Theta(n)$ height, and hence a splay applied to the lowest leaf will take $\Theta(n)$ time. However, the resulting splayed tree will have an average node depth roughly decreased by half!
- **Two quantities: real cost** and **increase in balance**
  - If we spend much, then we will also be balancing a lot
  - If don't balance a lot, than we also did not spend much
- A fully fledged formal proof of the efficiency is out ot the scope of this course (it involves the concept of **amortized analysis**)
  (if you are really curious you can for instance check the original paper)

Visualizing Splay Trees

- You can try the indicated url:

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



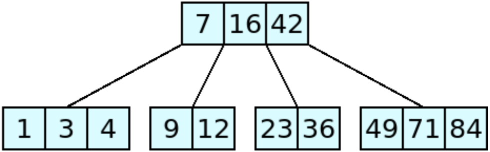
B-Trees

- 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")



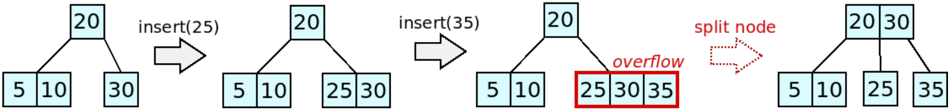
An example B-Tree of order 4

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

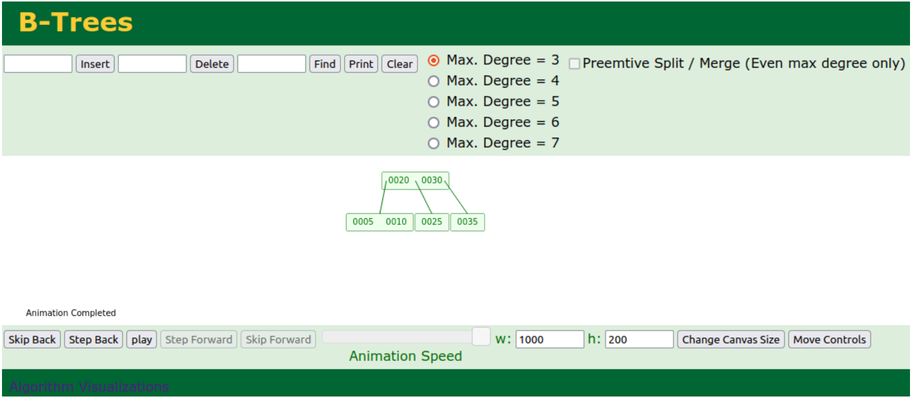
Example insertions in a B-Tree of order 3:



Visualizing B-Trees

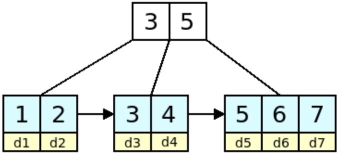
- You can try the indicated url:

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