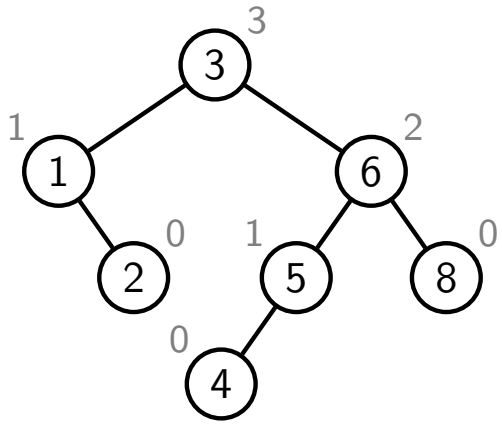
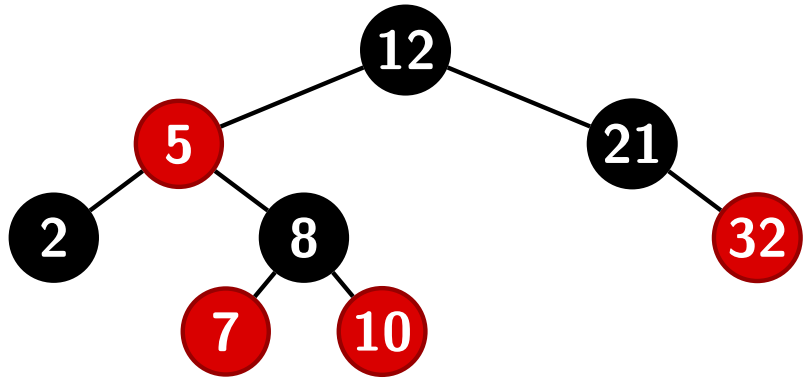
Balanced Binary Search Trees

L.EIC

Algorithms and Data Structures

2026/2027





P Ribeiro, AP Tomas

Binary Search Trees - Motivation

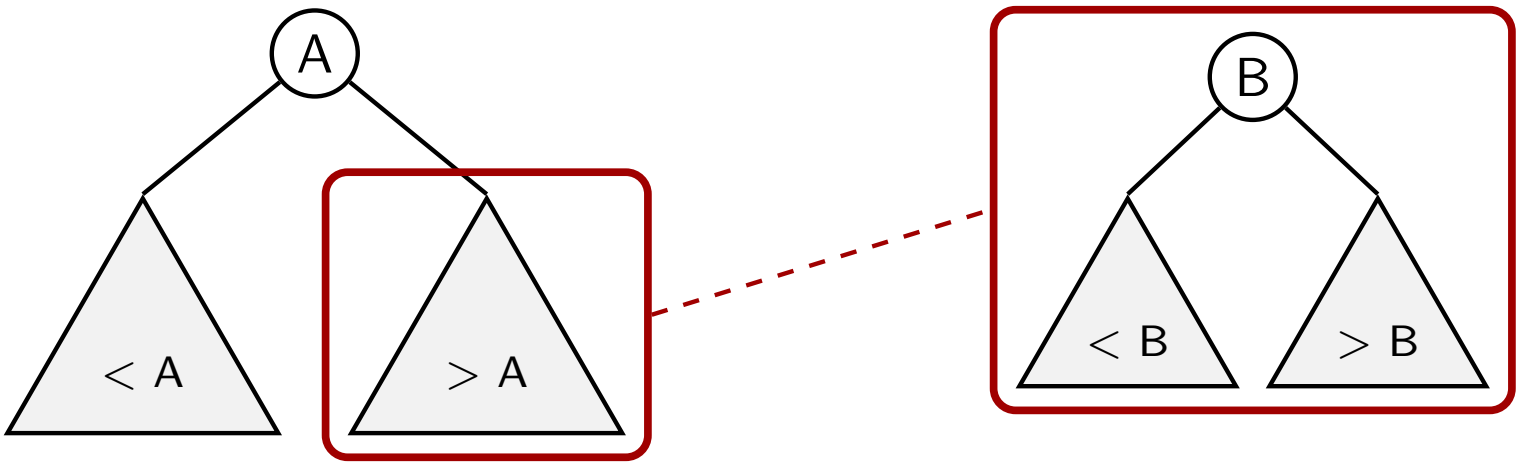
- Let S be a set of **"comparable"** objects/items:
  - Let a and b be two objects.
    They are "comparable" if it is possible to say whether $a < b$, $a = b$, or $a > b$.
  - An example could be numbers, but it could be something else (strings, students with a name and a student number; teams with points, goals scored and conceded, etc.)

- Some possible **problems** of interest:
  - Given a set S, determine if **a given item is in** S
  - Given a **dynamic set** S (which undergoes changes: additions and removals), determine if **a given item is in** S
  - Given a **dynamic set** S, determine the **largest/smallest** item in S
  - **Sort** a set S
  - ...

- **Binary Search Trees!**

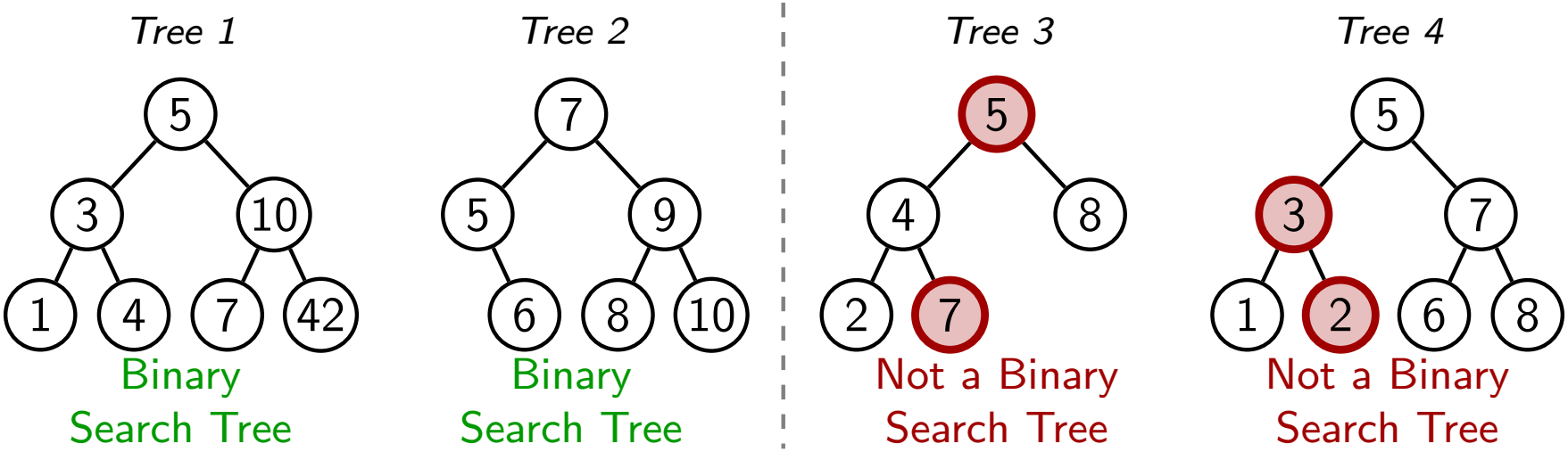
Binary Search Trees - Concept

- For **every** node in the tree, the following must happen:
  the node is greater than all the nodes in its left subtree and smaller than all the nodes in its right subtree



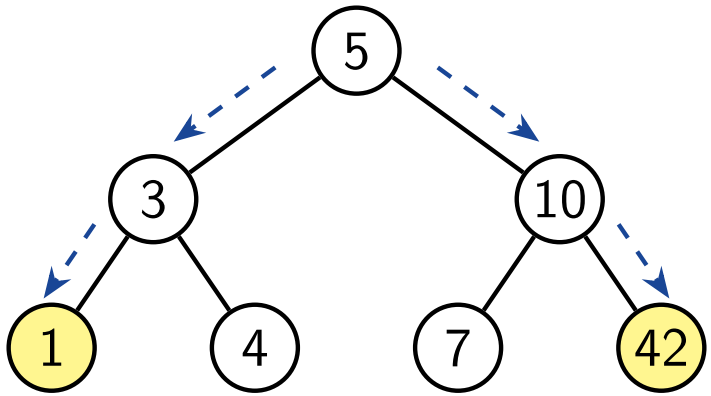
Binary Search Trees - Examples

- For **every** node in the tree, the following must happen:
  the node is greater than all the nodes in its left subtree and smaller than all the nodes in its right subtree



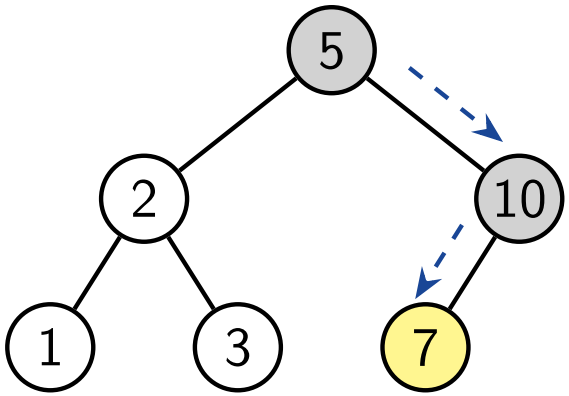
- In trees 1 and 2, the conditions are respected
- In tree 3, node 7 is on the left of node 5, but $7 > 5$
- In tree 4, node 2 is on the right of node 3, but $2 < 3$

Binary Search Trees - Some Consequences



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

Binary Search Trees - Searching for a Value



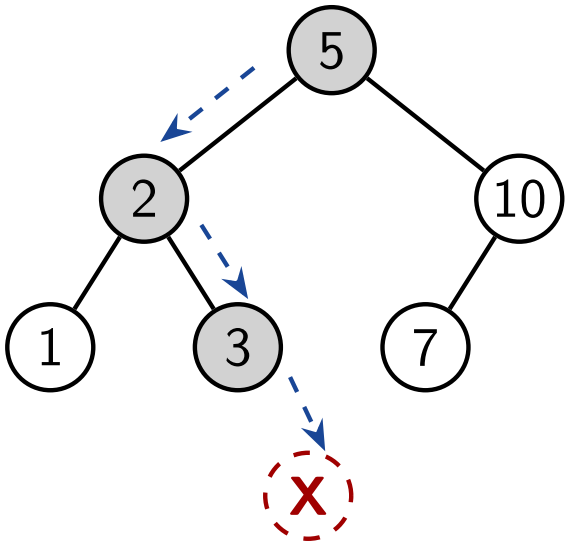
contains(7) ?

$7 = 7$: found!

true

- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node

Binary Search Trees - Searching for a Value

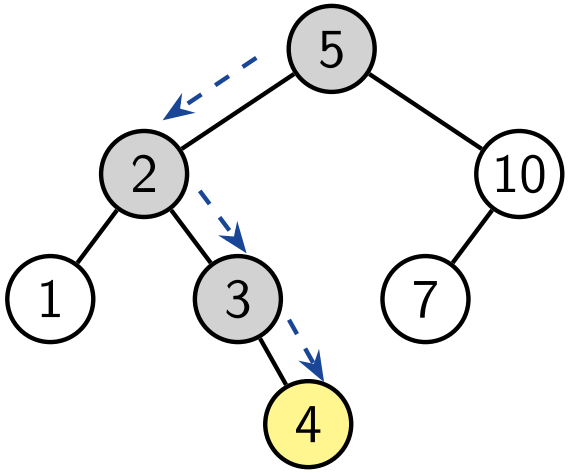


contains(4) ?

empty subtree: not found
false

- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node

Binary Search Trees - Inserting a Value



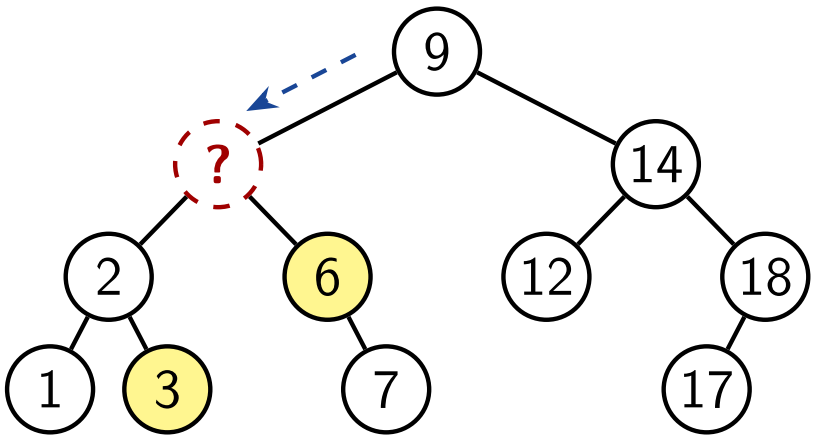
insert(4)

empty subtree:
insert 4 here

- Start at the root, and traverse the tree
- Choose the left or right branch based on whether the value is smaller or greater than the "current" node
- Insert at the corresponding leaf position

Note: If the value is equal to an already existing one, it is typically *not inserted*. If we want to allow repeated values (a *multiset*), we must be consistent and always choose a position (e.g., always insert on the left, where left subtree nodes would be $\leq$ and right subtree nodes would be $>$)

Binary Search Trees - Removing a Value



remove(5)

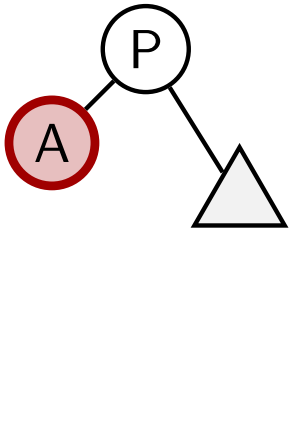
candidates: 3 or 6

- Start at the root, and traverse the tree until the value is found
- Once the value is found, what should we do next?
  - If the node to be removed is a leaf (no children), simply remove it
  - If it has only one child, simply "lift" that child to the corresponding position
  - If it has two children, the candidates to replace it are:
    - ★ The largest node in the left branch, or
    - ★ The smallest node in the right branch

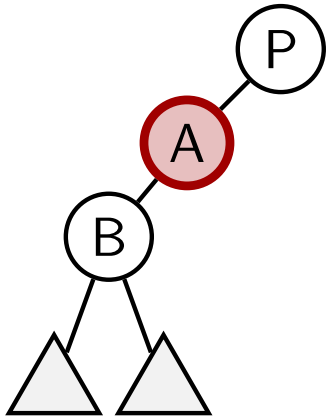
Binary Search Trees - Removing a Value

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

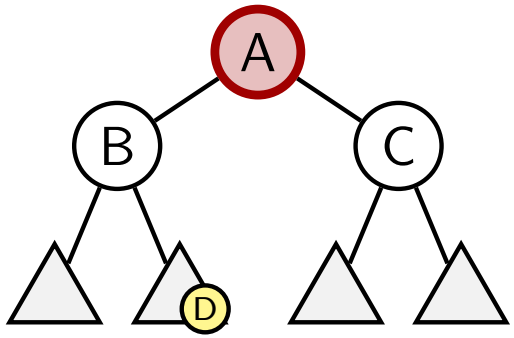
Leaf

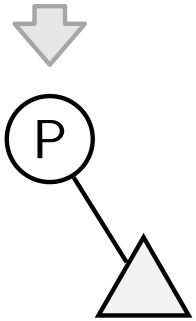


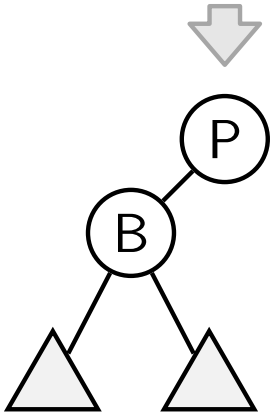
One child
(left or right)

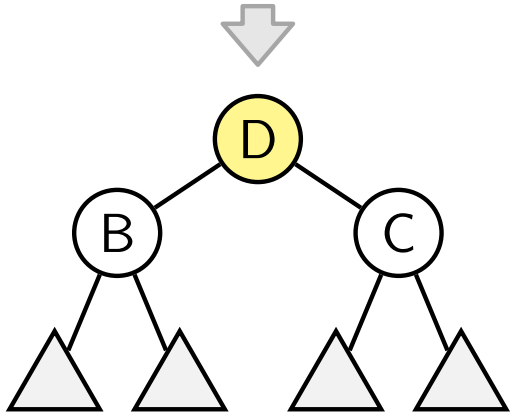


Two children



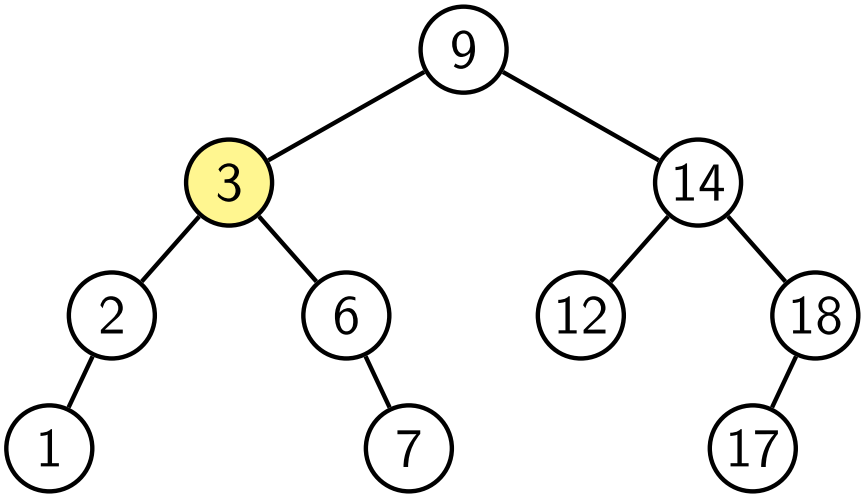






Binary Search Trees - Removing a Value

Example with two children

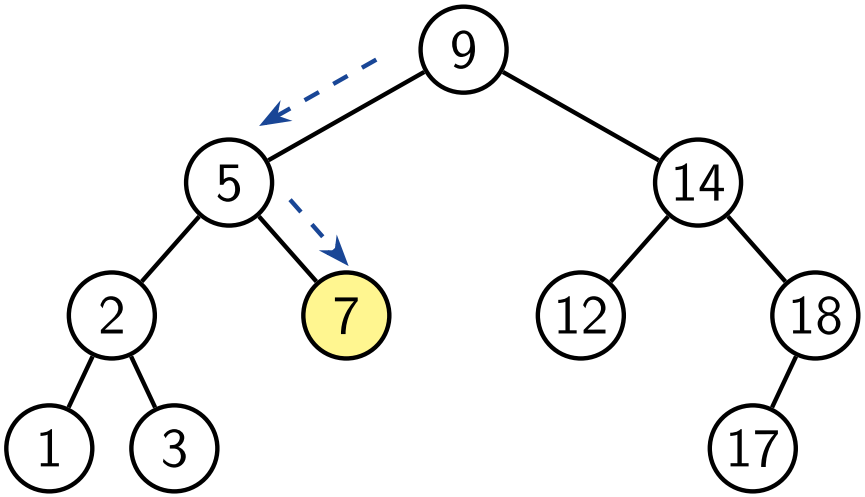


remove(5)

copy 3 into the place of 5
and remove the old 3
(a leaf)

Binary Search Trees - Removing a Value

Example with only one child



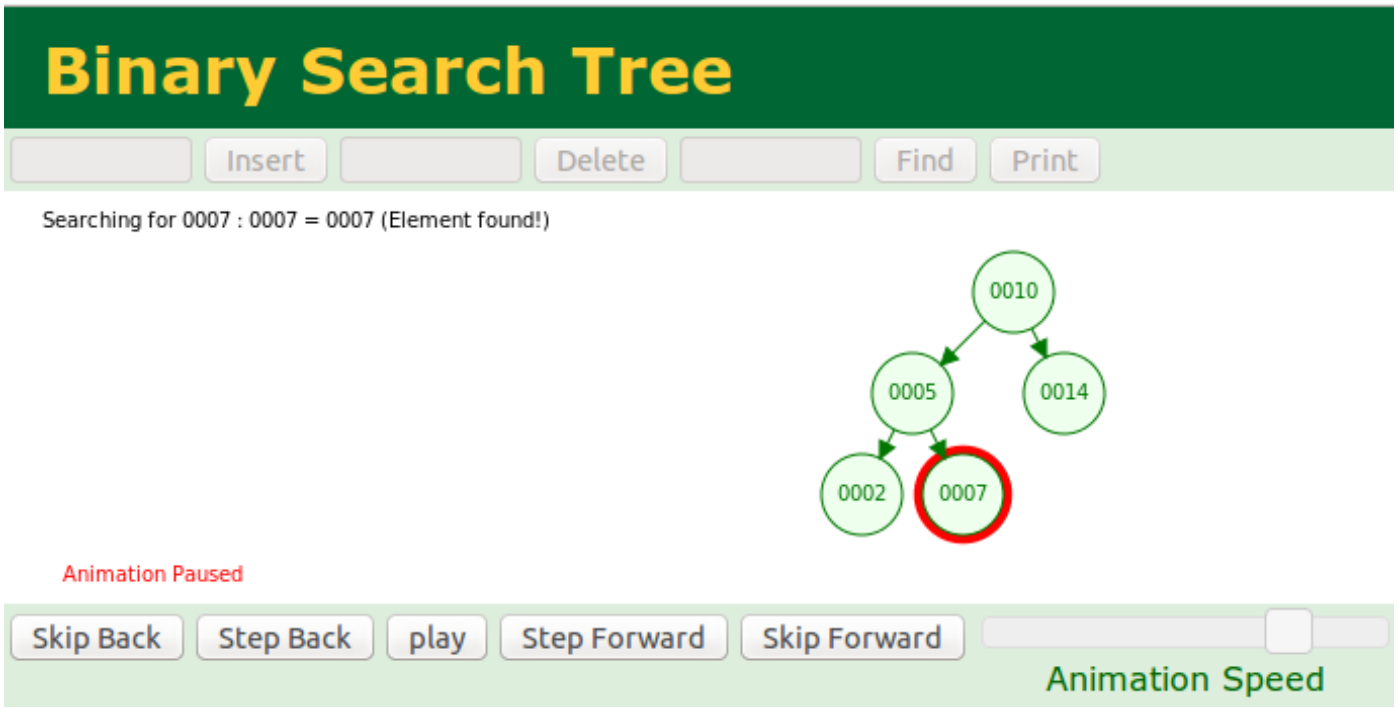
remove(6)

lift 7 to the place of 6

Binary Search Trees - Visualization

- You can visualize search, insertion, and removal (try the provided URL):

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



Binary Search Trees - An Implementation

- We are not going to analyze in detail during the lecture, but here is a possible implementation of an (unbalanced) binary search tree
- The core of the class is exactly the same as before.

```
template <class T> class BSTree {
private:
  struct Node {
    T value;
    Node *left, *right;
  };

  Node *root;

public:
  BSTree() {root = nullptr;}

  // ...
}
```

Binary Search Trees - Implementation

- What changes are the other methods, such as **contains**, **insert**, and **remove**, which can take advantage of the fact that it is a binary search tree.
- These are methods of the BSTree class. For a visual explanation of what they do, you can refer to previous slides (slides 6 to 12).
- Let's start with **contains**.

```
// Is the value contained in the tree?
bool contains(const T & val) const {
  return contains(root, val);
}

bool contains(Node *n, const T & val) const {
  if (n == nullptr) return false;
  if (val < n->value) return contains(n->left, val);
  if (val > n->value) return contains(n->right, val);
  return true;
}
```

Binary Search Trees - Implementation

- For **insert**, we take advantage of the return value of a recursive function.

```
// Add an element to a search tree
// Returns true if insertion succeeded, false otherwise
bool insert(const T & val) {
  if (contains(val)) return false;
  root = insert(root, val);
  return true;
}

Node *insert(Node *n, const T & val) {
  if (n == nullptr) {
    Node *aux = new Node;
    aux->value = val;
    aux->left = aux->right = nullptr;
    return aux;
  } else if (val < n->value) {
    n->left = insert(n->left, val);
  } else if (val > n->value) {
    n->right = insert(n->right, val);
  }
  return n;
}
```

Binary Search Trees - Implementation

- **remove**: replace removed value with largest value from left subtree.

```
// Remove an element from a search tree
// Returns true if removal succeeded, false otherwise
bool remove(const T & val) {
  if (!contains(val)) return false;
  root = remove(root, val);
  return true;
}

Node *remove(Node *n, const T & val) {
  if (val < n->value) n->left = remove(n->left, val);
  else if (val > n->value) n->right = remove(n->right, val);
  else if (n->left == nullptr) {
    Node *tmp = n->right; delete n; return tmp; // free the removed node
  } else if (n->right == nullptr) {
    Node *tmp = n->left; delete n; return tmp;  // free the removed node
  } else {
    Node *max = n->left;
    while (max->right != nullptr) max = max->right;
    n->value = max->value;
    n->left = remove(n->left, max->value);
  }
  return n;
}
```

Binary Search Trees - Test

- An example of usage (assuming the other methods were implemented as in normal Binary Trees):

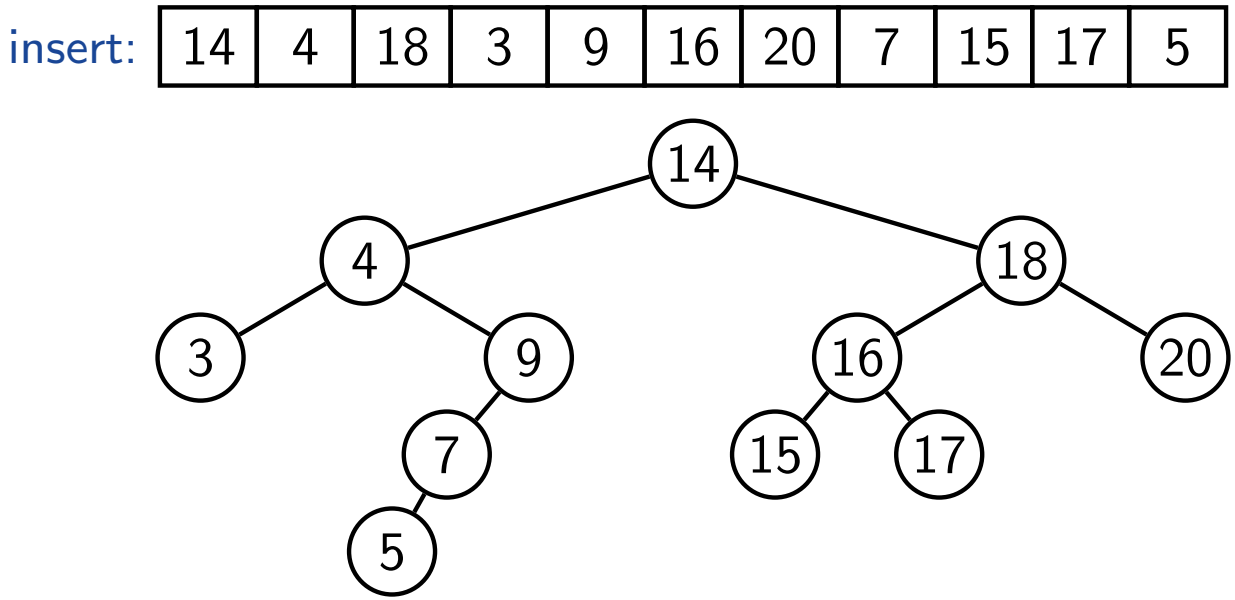
```
#include "binarySearchTree.h"

int main() {
  // Tree creation
  BSTree<int> t;
  // Inserting 11 elements in the binary search tree
  int data[] = {14, 4, 18, 3, 9, 16, 20, 7, 15, 17, 5};
  for (int i=0; i<11; i++) t.insert(data[i]);

  // Writing the result of calling some methods
  std::cout << "numberNodes = " << t.numberNodes() << std::endl;
  std::cout << "height = " << t.height() << std::endl;
  std::cout << "contains(2) = " << t.contains(2) << std::endl;
  std::cout << "contains(3) = " << t.contains(3) << std::endl;
  // Writing the nodes of the tree following several possible orders
  t.printPreOrder(); t.printInOrder(); t.printPostOrder();
  // Trying node removal
  t.remove(14);
  t.printPreOrder(); t.printInOrder(); t.printPostOrder();
}
```

Binary Search Trees (BST) - Test

- The code from the previous slide creates the following BST:



- Inorder, this tree becomes: `3 4 5 7 9 14 15 16 17 18 20`

When printed *inOrder*, the values of a BST are sorted in ascending order.

Binary Search Trees - Complexity

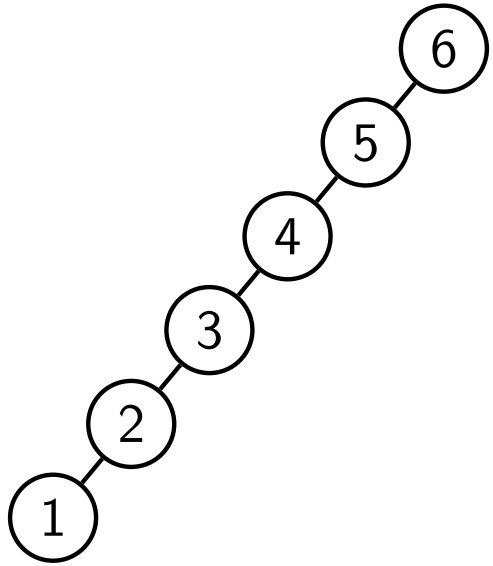
- How to characterize the **time each operation takes**?
  - All operations search for a node by traversing the **height** of the tree

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 performing a search, insertion, or removal in T is $\mathcal{O}(h)$.

Imbalance in a Binary Search Tree

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



inserting 6, 5, 4, 3, 2, 1 in this order (height $n - 1$)

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

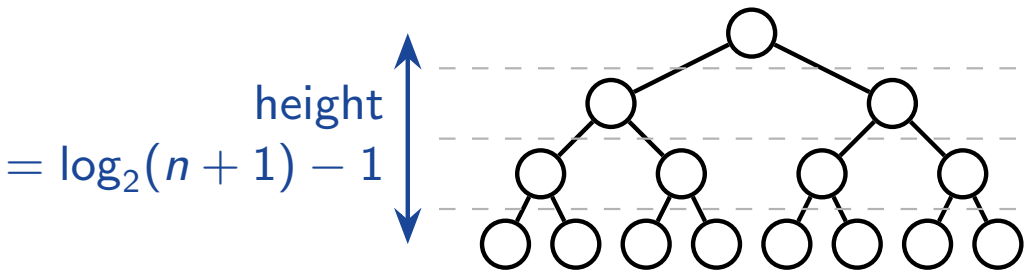
(The height depends on the order of insertion, and there are "bad" orders)

Balanced Trees

- We want trees... **balanced**



- In a balanced tree with n nodes, the height is ... $\mathcal{O}(\log n)$



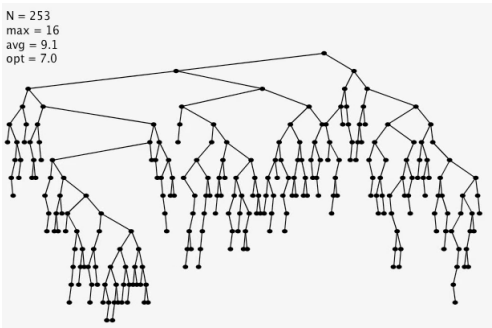
Nodes in level	Total nodes
$1 = 2^0$	$1 = 2^1 - 1$
$2 = 2^1$	$3 = 2^2 - 1$
$4 = 2^2$	$7 = 2^3 - 1$
$8 = 2^3$	$15 = 2^4 - 1$

Height for Random Order

Height of a tree with random elements

If we insert n elements in a completely random order into a binary search tree, its *expected height* is $\mathcal{O}(\log n)$, the same order as a balanced tree

If you're curious about a proof, check out the excellent book *"Introduction to Algorithms"* (not required for the exam)

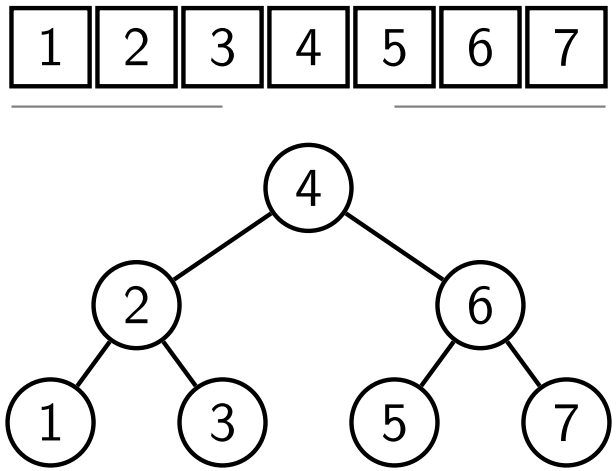


- In practice, the order is often **not random**: data may arrive (almost) sorted, e.g. dates, student numbers or names, giving a chain-like tree with height close to $n - 1$
- Removals also unbalance the tree over time
- So a random order is **not a guarantee**: how can we make sure the tree is **always** balanced?

Balanced Trees

For a given set of numbers, **which order to insert** in a binary search tree so that it becomes as balanced as possible?

Answer: *"binary search"* - if the numbers are sorted, insert the middle element, split the remaining list at that element, and insert the remaining elements from each half in the same manner.



insertion order: 4, 2, 6, 1, 3, 5, 7

Balancing Strategies

- If we know all the elements in advance, we can insert them in a good order (previous slide), or simply shuffle them first
- But what if we need to **dynamically** insert and remove elements?
- 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
- Red-Black Trees
- Splay Trees
- B-Trees
- Treaps, ...

Other data structures:

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

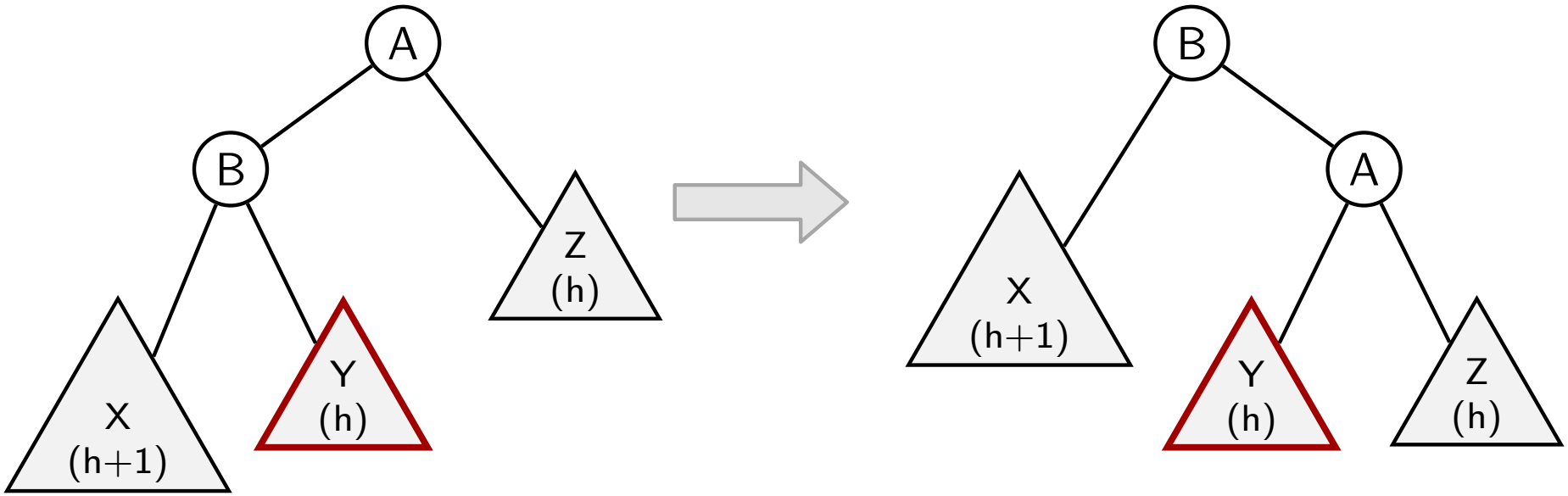
What's next?

- Plan for the rest of today:

  1. **Rotations**: the basic tool for rebalancing
  2. **AVL Trees**
  3. **Red-Black Trees**
  4. BSTs in the STL: **set** and **map** (and also **multiset** and **multimap**), with live coding on real data
  5. A quick look at other trees: **splay trees** and **B-trees**

Rotations

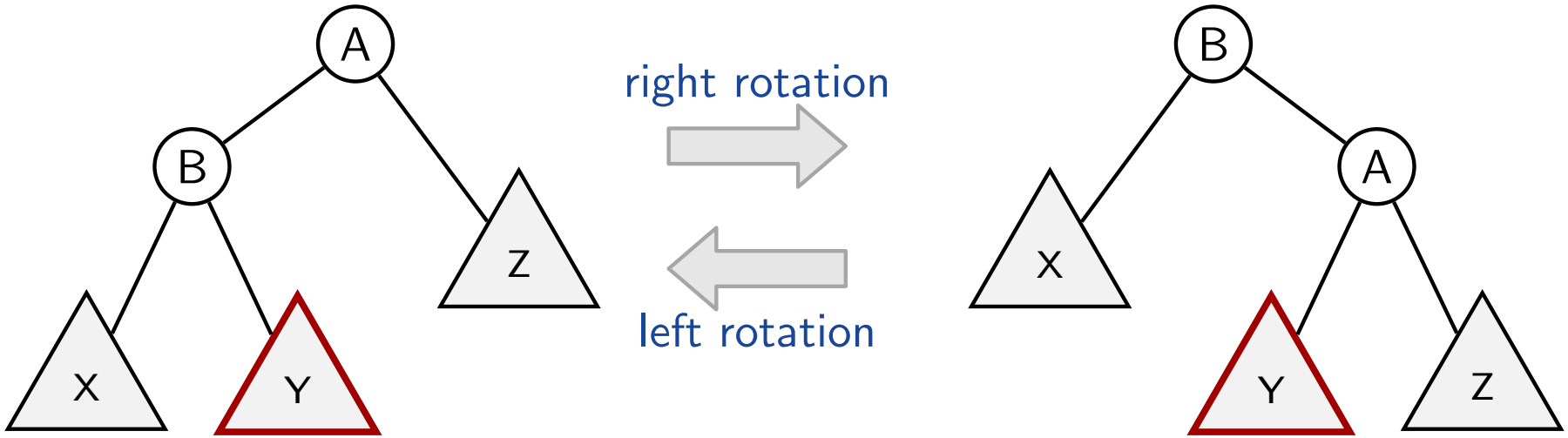
- Simple case: **how to balance** the following tree (in parentheses: the height of each subtree):



This operation is called a **right rotation**

Rotations

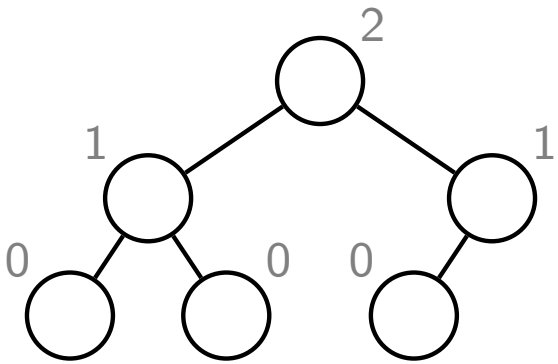
- The relevant rotation operations are the following:
  - Note that we must not break the properties that turn the tree into a binary search tree



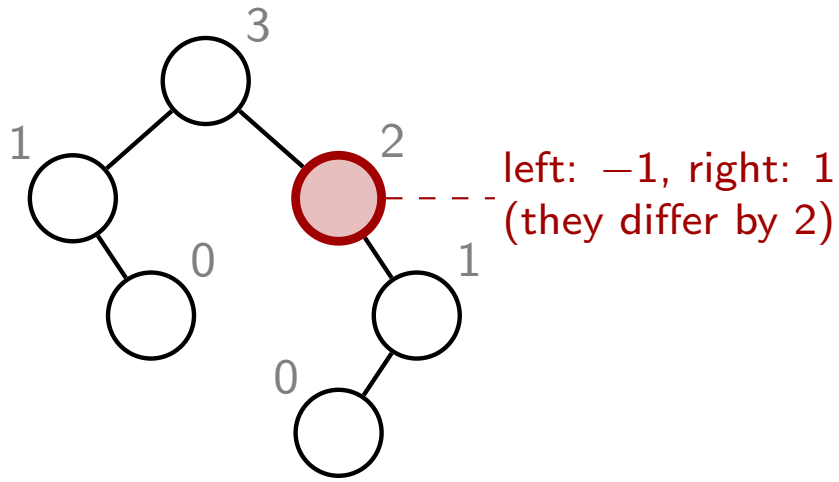
AVL Trees

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

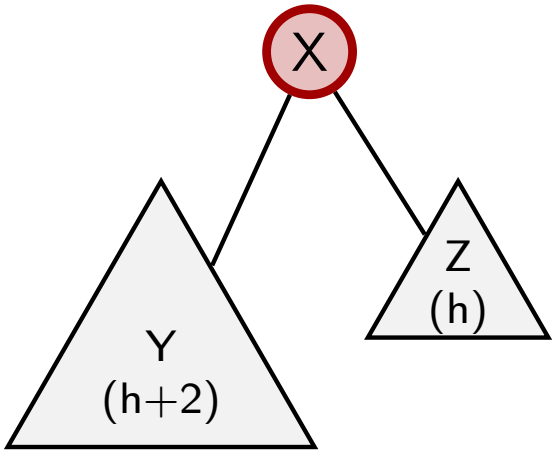


Not an AVL Tree

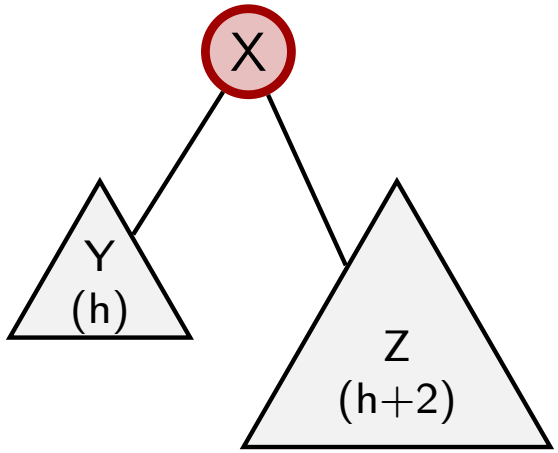
- When inserting and removing nodes, we change the tree so that we keep the **height invariant**

AVL Trees

- **Inserting** on an 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

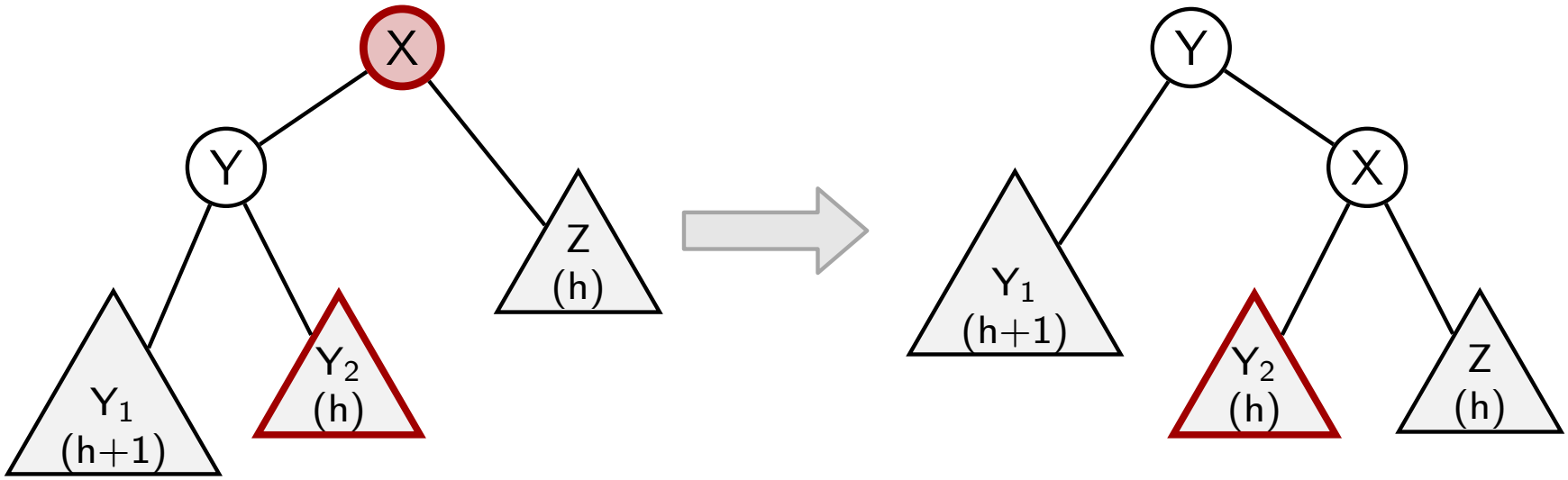


+2 on the right

- 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:



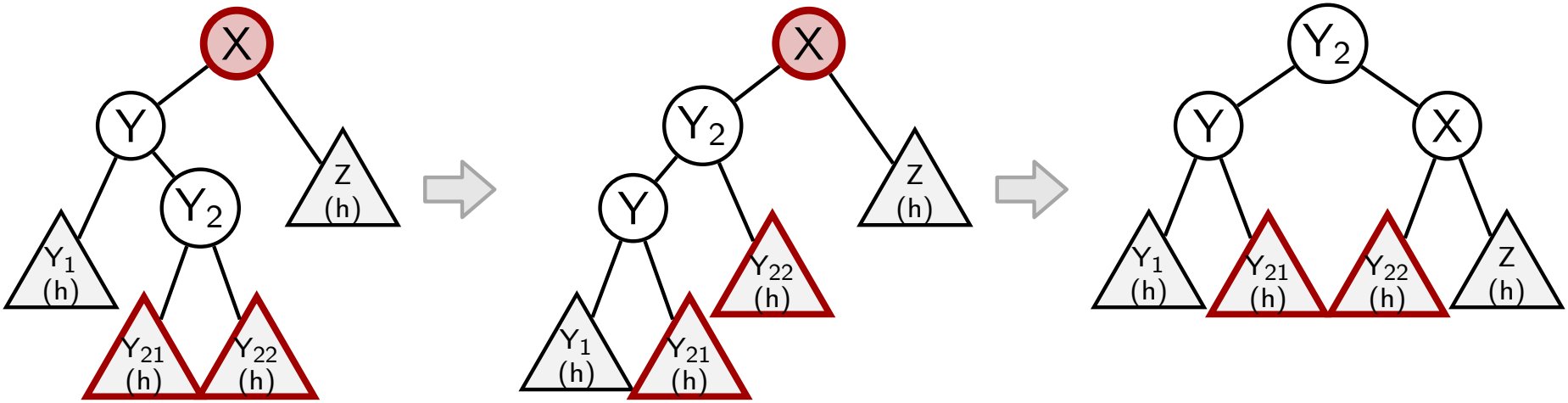
Left is too "heavy", case 1

We correct by making a right rotation starting in X

- Note: Y_2 might also have height $h+1$ (this can happen after a removal): 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

To summarize:

- 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

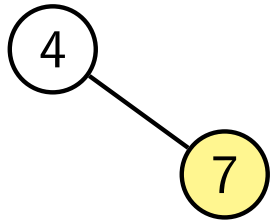
AVL Trees

- **Example** of node insertion:



AVL Trees

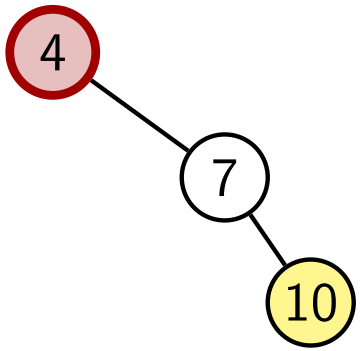
- **Example** of node insertion:



insert(7)

AVL Trees

- **Example** of node insertion:

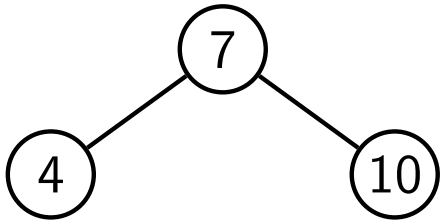


insert(10)

4 is unbalanced
(+2 on the right)

AVL Trees

- **Example** of node insertion:

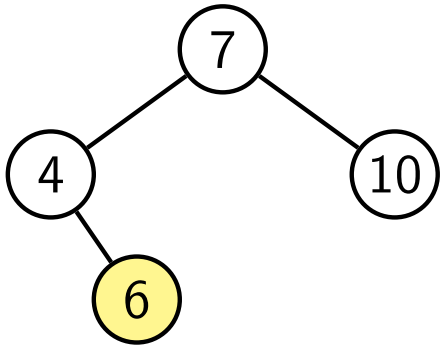


insert(10)

left rotation at 4

AVL Trees

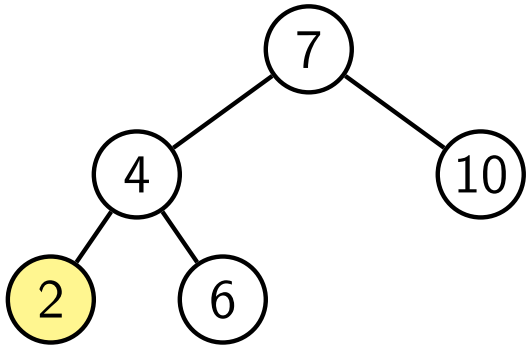
- **Example** of node insertion:



insert(6)

AVL Trees

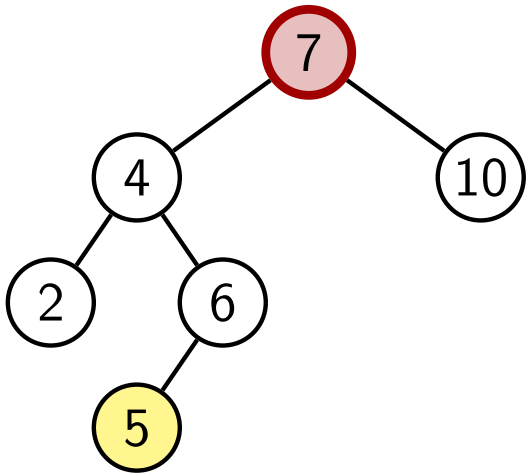
- **Example** of node insertion:



insert(2)

AVL Trees

- **Example** of node insertion:

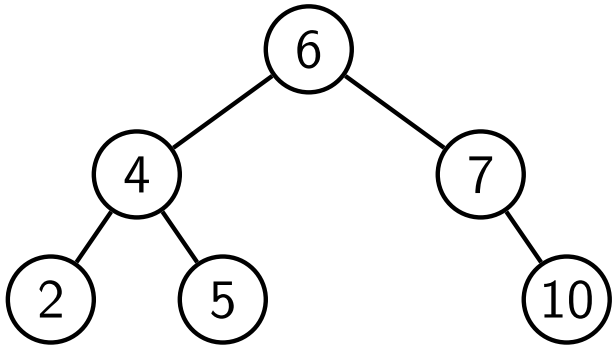


insert(5)

7 is unbalanced
(+2 on the left, case 2)

AVL Trees

- **Example** of node insertion:



insert(5)

after two rotations:
left at 4, right at 7

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 then 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?

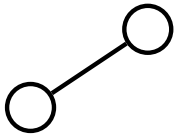
AVL Trees

- 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?
  - Let $N(h)$ be the **minimum** number of nodes of an AVL tree with height h
- **Why the smallest?** We flip the question: instead of *"with n nodes, how tall can an AVL tree get?"*, we ask *"to reach height h, how few nodes are enough?"*
  - The smallest tree of height h is the **tallest possible** tree for its number of nodes
  - If even the smallest tree of height h needs $N(h)$ nodes, a tree with n nodes can only reach height h if $n \geq N(h)$

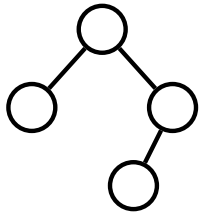
AVL Trees



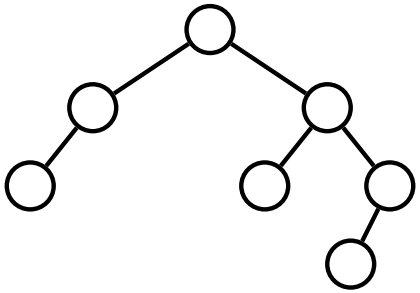
Height 0



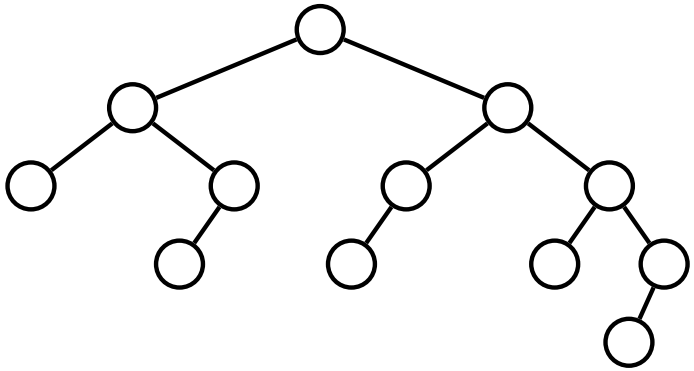
Height 1



Height 2



Height 3



Height 4

AVL Trees

- Summarizing:
  - $N(0) = 1$
  - $N(1) = 2$
  - $N(2) = 4$
  - $N(3) = 7$
  - $N(4) = 12$
  - ...
  - $N(h) = N(h-2) + N(h-1) + 1$
- 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
  - $\log(N(h)) \approx \log(\phi)h$
  - $h \approx \frac{1}{\log(\phi)} \log(N(h))$
  - Since $n \geq N(h) \geq \phi^h$: $h \leq \log_\phi n = \frac{1}{\log_2 \phi} \log_2 n \approx 1.44 \log_2 n$

The height h of an AVL tree with n nodes is roughly (at most) $1.44 \log_2 n$, which is $\mathcal{O}(\log n)$

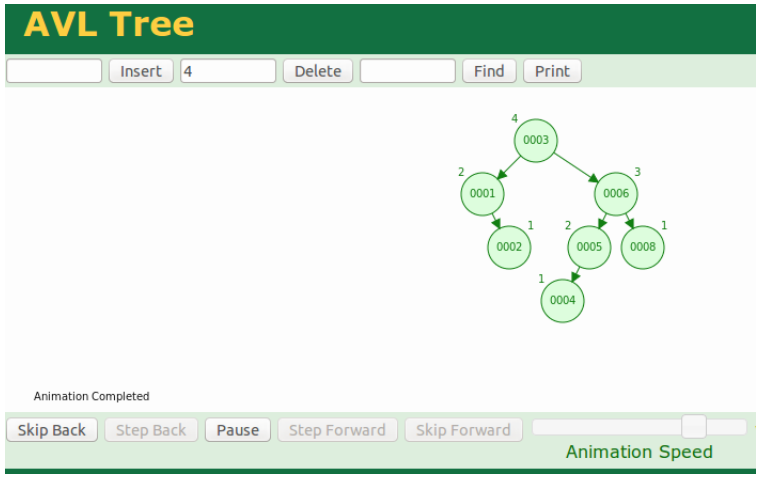
AVL Trees

- **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 $1.44 \log_2 n$ 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

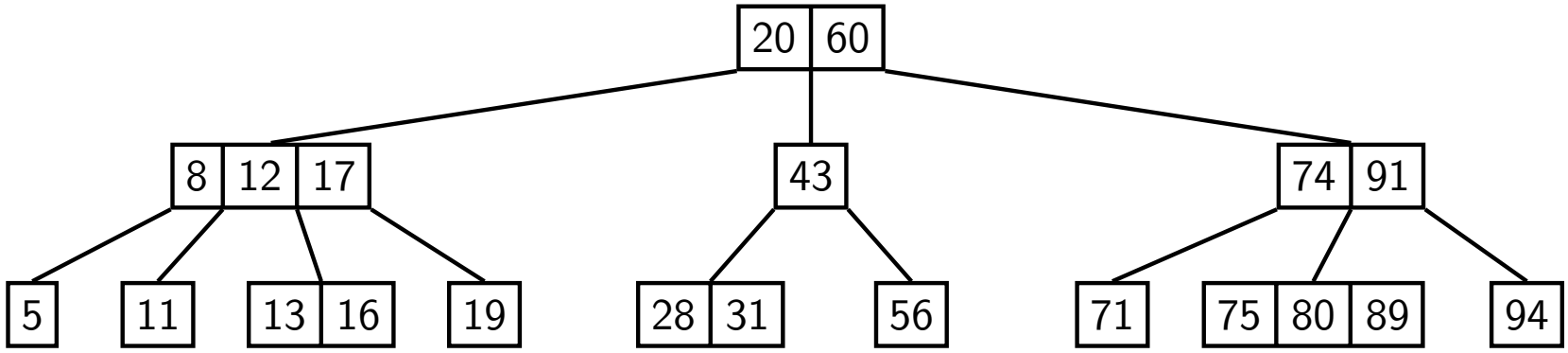
- The name AVL comes from the authors: G. **A**delson-**V**elsky and E. **L**andis. The original paper describing them is from 1962 (*"An algorithm for the organization of information"*, Proceedings of the USSR Academy of Sciences)

- 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



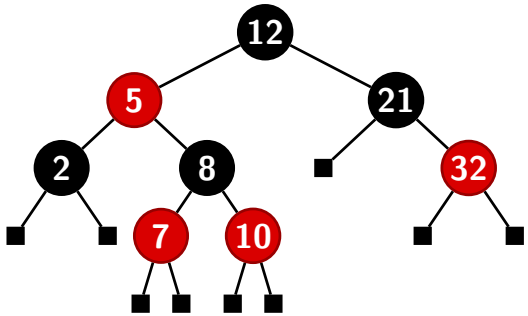
- The original paper is from 1978 and it was written by L. Guibas and R. Sedgewick (*"A Dichromatic Framework for Balanced Trees"*)
- The authors say they use the red and black colors because they looked good when printed and because those were the pen colors that they had available to draw the trees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

Red-Black Trees

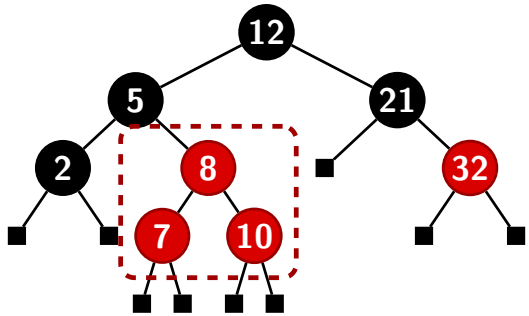
Red-Black Tree

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

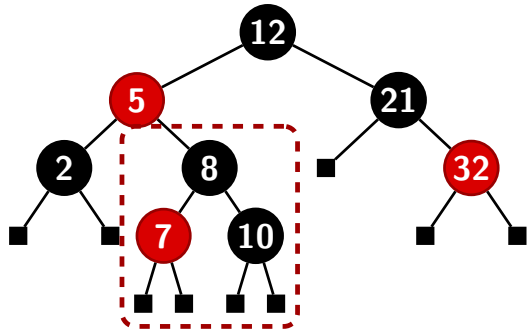
- **(root property)** The root node is black
- **(leaf property)** The leaves are null/empty black nodes
- **(red property)** The children of a red node are black
- **(black property)** For each node, a path to any of its descending leaves has the same number of black nodes



Red-Black Tree



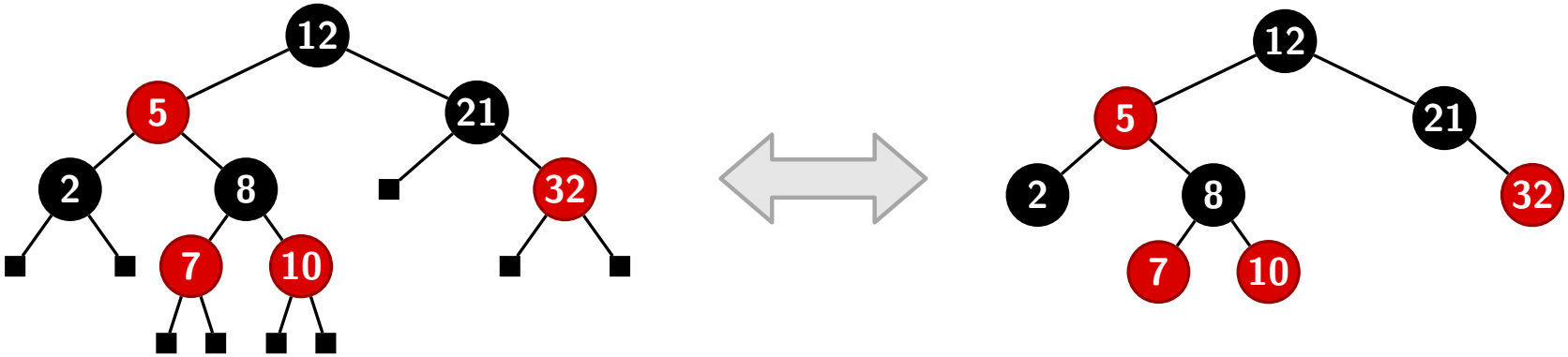
Not a Red-Black Tree
(breaks the red property)



Not a Red-Black Tree
(breaks the black property)

Red-Black Trees

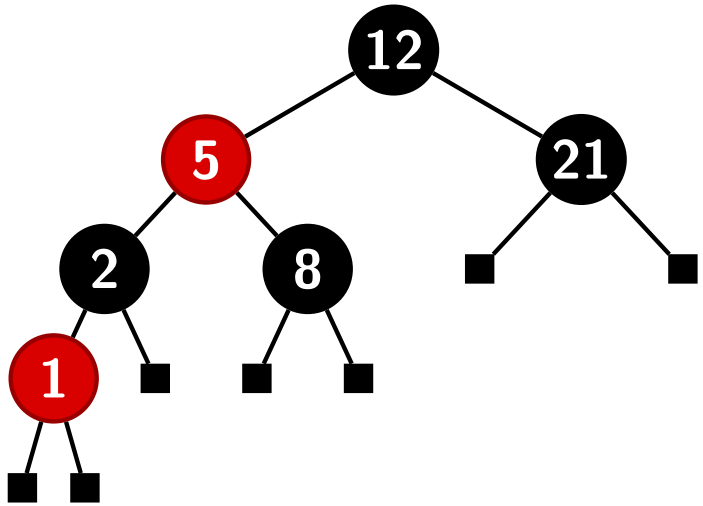
- For better visibility, the images may not contain the "null" leaves, but you may assume those nodes exist.
  We call the non-null nodes **internal nodes**.



- 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



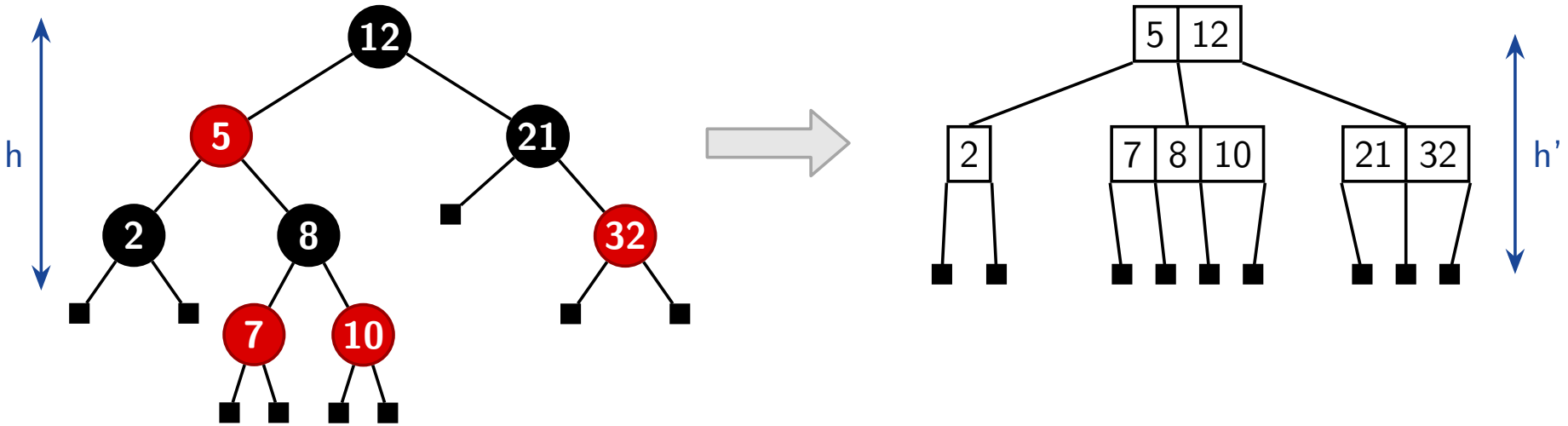
Red-Black Trees

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:

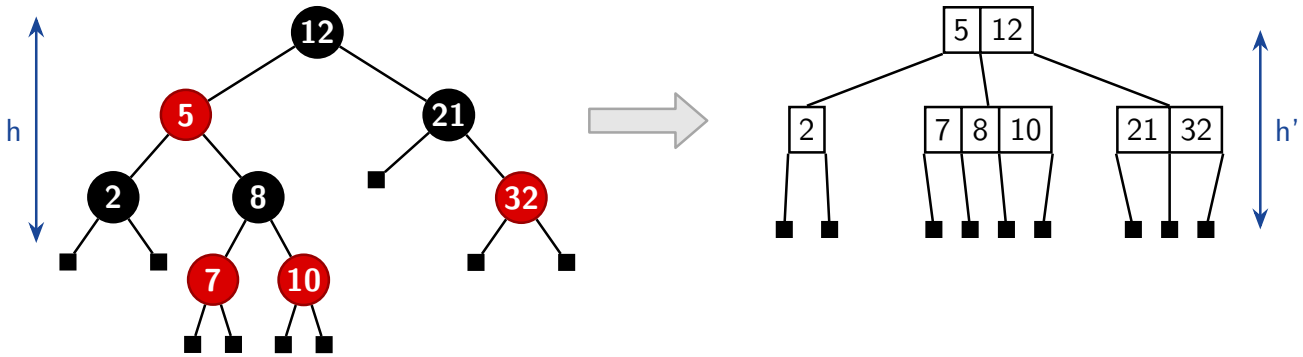


- This process produces a tree with 2, 3 or 4 children
- This 2-3-4 tree has leaves at a uniform height of **h'** (where h' is the *black height*)

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: $n \geq 2^{h'} - 1$
- $n + 1 \geq 2^{h'}$
- $\log_2(n+1) \geq h' \geq h/2$
- $h \leq 2\log_2(n+1)$ □

Red-Black Trees - A quick recap

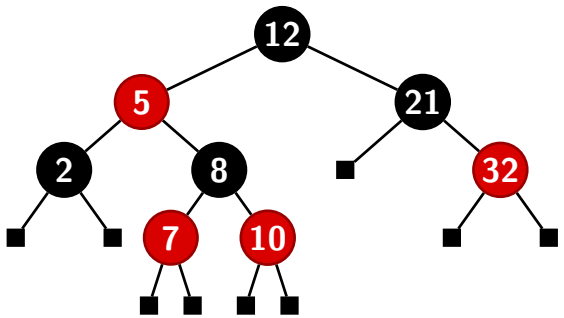
Red-Black Tree

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 $h \leq 2 \times \log_2(n+1)$
[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

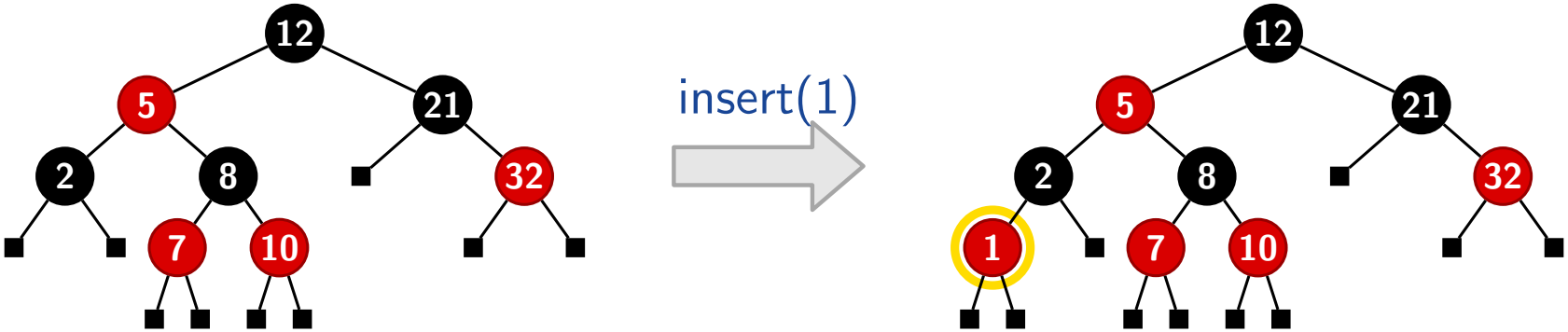
- Insert as in any binary search tree
- Color the inserted node as red (adding the null black nodes)
- Recolor and restructure if needed (restore the invariants)

- 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 that 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:

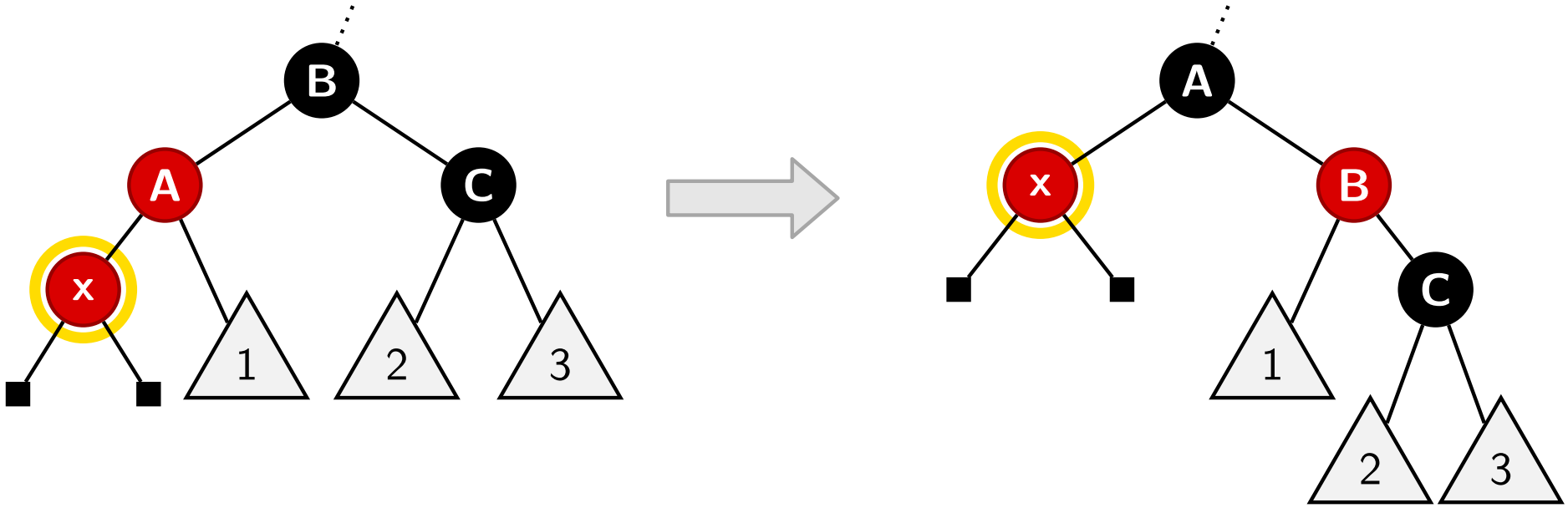
Example:



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

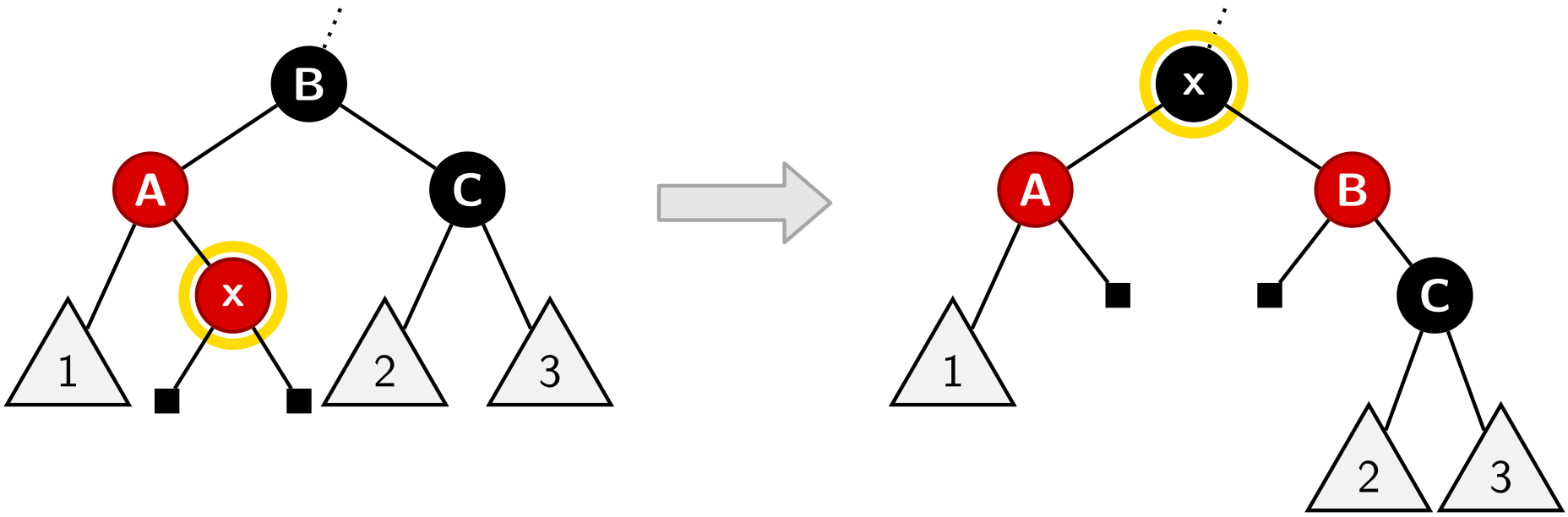


Description: right rotate the grandparent, followed by swapping the colors between the parent and the grandparent

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



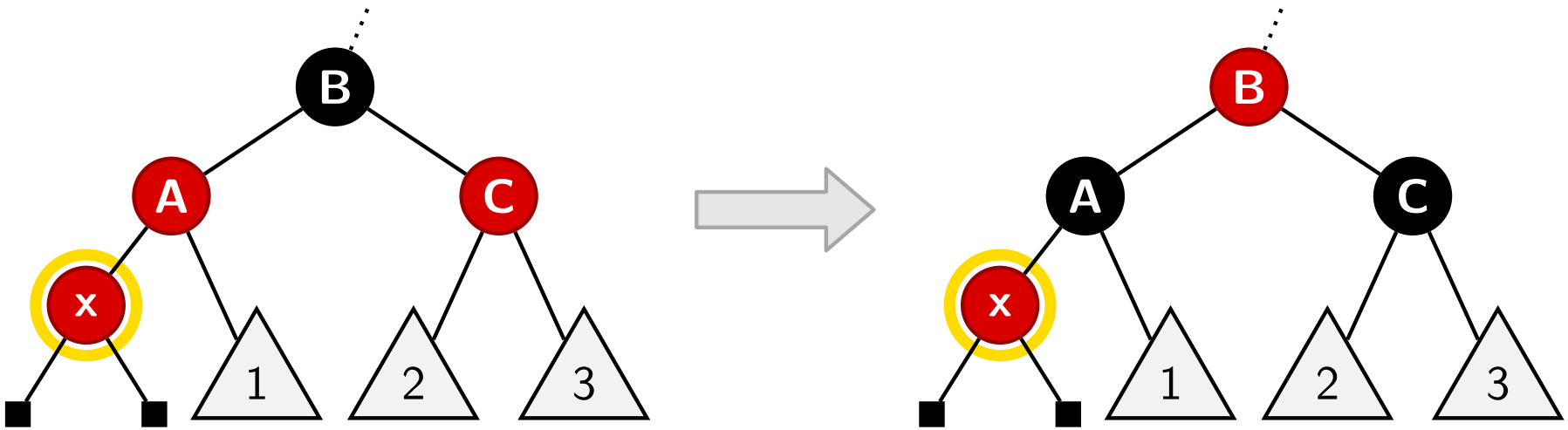
Description: left rotation of parent followed by the moves of 1.a

[If the parent was the right child of the grandparent, 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



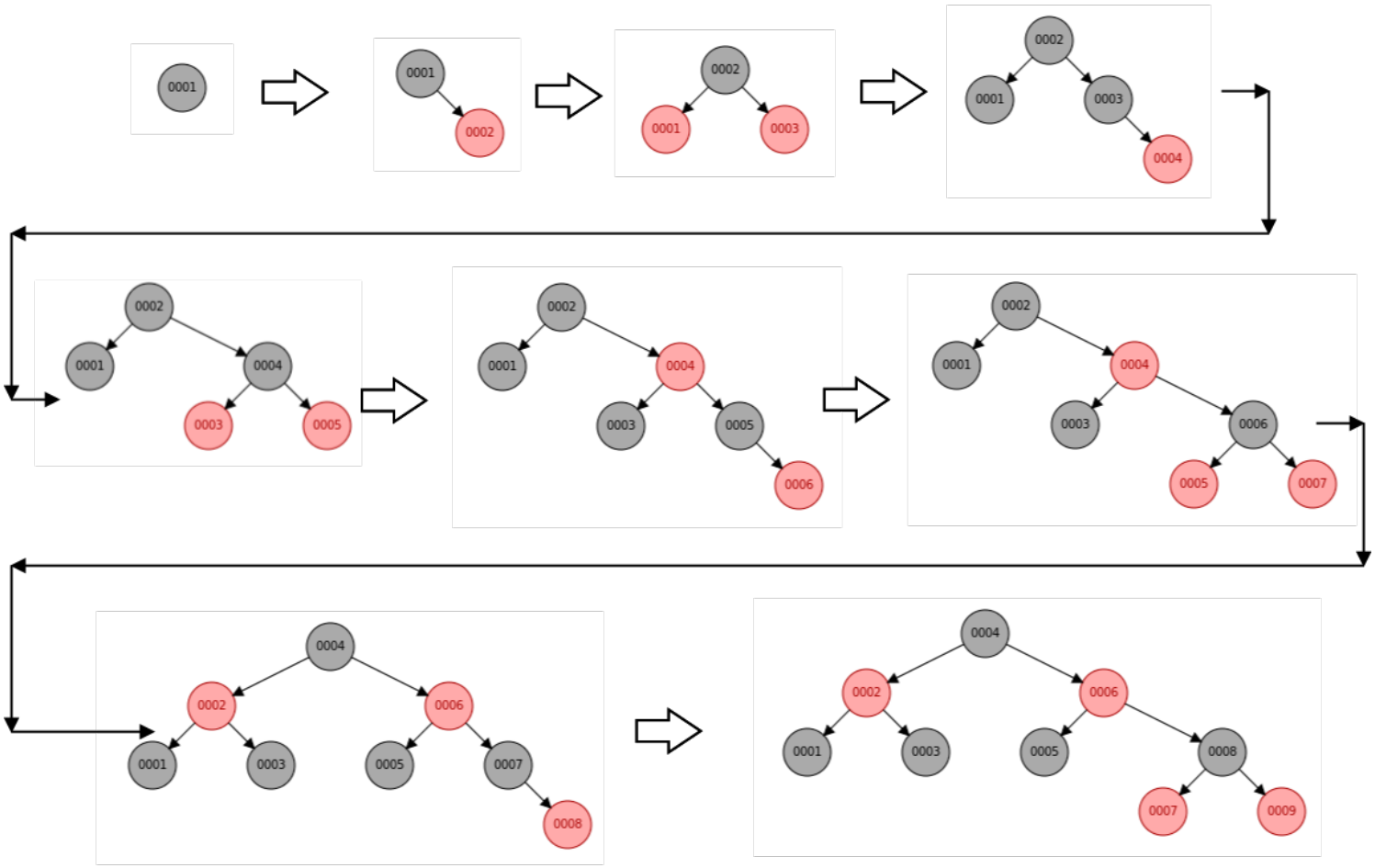
Description: swap colors of parent, uncle and grandparent

Now, if the parent of the grandparent 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
  - At most $\mathcal{O}(\log n)$ recolorings (case 2 may repeat up the tree) and at most 2 rotations

- 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 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 $\sim 2 \log_2 n$ *vs* AVL with height $\sim 1.44 \log_2 n$
  - 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 by the most common C++ standard library implementations** 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.
- Recent standards: C++14, C++17, C++20, C++23 and the recently finalized C++26
- 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

  There are also *unordered* versions (`unordered_set`, `unordered_map`, ...), implemented with hash tables *(in another class)*

- Usual operations are available:
  - Iterators (forward and reverse)
  - Lookup (find, count, lower_bound, upper_bound, ...)
  - Modifiers (clear, insert, erase, ...)

Example Applications

LIVE

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

```
Jose
Goncalo
Antonio
Leonor
...
```

```
// 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

LIVE

- 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 into the set (keys are 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 445 names
```

Example Applications

LIVE

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;
}
```

```
Adriana
Afonso
Aires
...
```

Example Applications

LIVE

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

- Using iterators (`begin()` in $\mathcal{O}(1)$; increment and decrement in $\mathcal{O}(1)$ amortized, $\mathcal{O}(\log n)$ in the worst case):

```
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: Adriana
2nd name: Afonso
3rd name: Aires
2nd name: Afonso
Last name: Zoe
Second to last name: Yili
```

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

Example Applications

LIVE

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 = 30
```

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

Example Applications

LIVE

- What if we want the frequency of each name?

```
map<string, int> m; // maps a name to its frequency
string name;
while (cin >> name) {
  m[name]++; // if name is new, m[name] is created with value 0
}

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

```
Adriana 1
Afonso 15
Aires 1
...
```

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

Example Applications

LIVE

- 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;
```

```
Da
Ana
Jose
Pedro
Leonor
...
```

- Careful: two names with the same length are now considered *equal*, so only one name of each length is kept!

Example Applications

LIVE

- 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;}
};

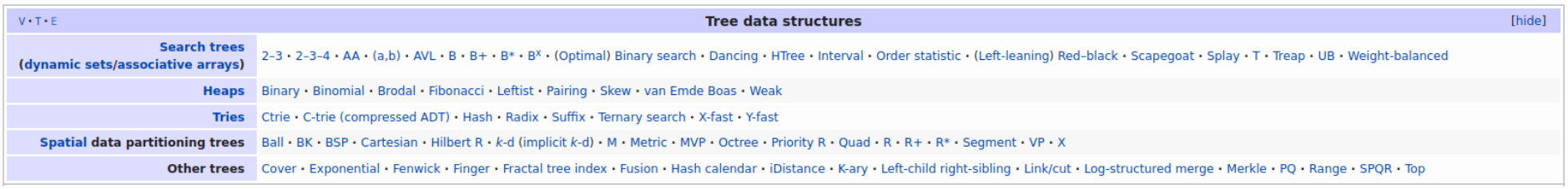
bool operator< (const Person & p1, const Person & p2) {
  return p1.surname < p2.surname;
}
```

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

```
Patrick Daniel
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.



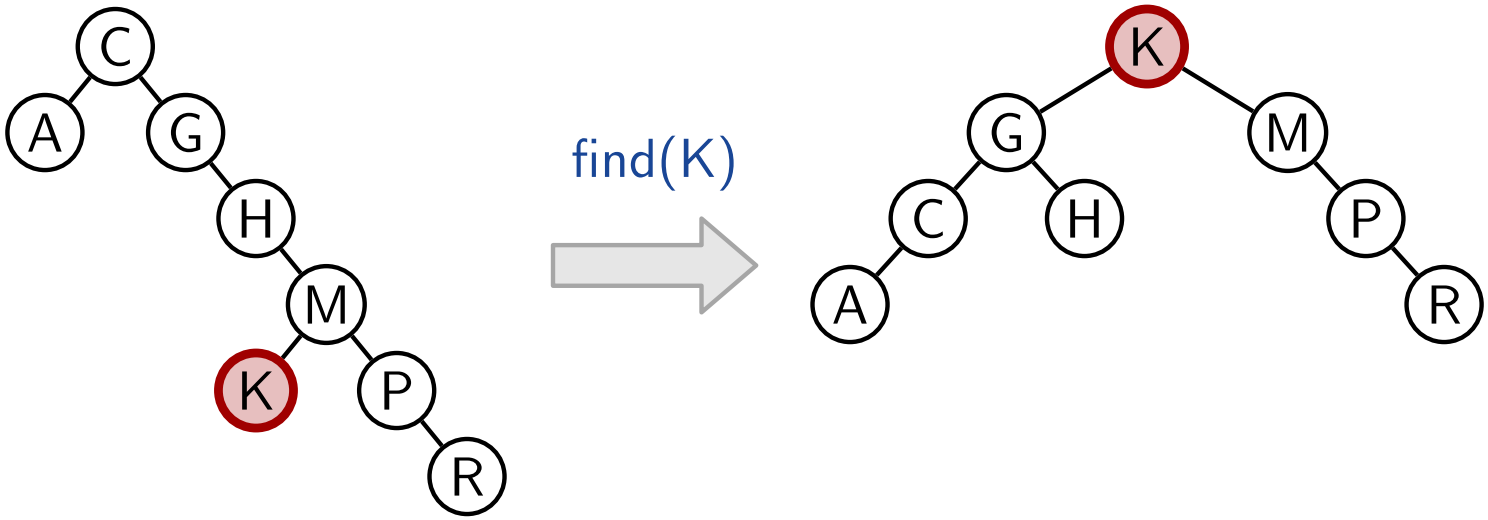
V • T • E	Tree data structures	[hide]
Search trees (dynamic sets/associative arrays)	2–3 • 2–3–4 • AA • (a,b) • AVL • B • B+ • B* • B^x • (Optimal) Binary search • Dancing • HTree • Interval • Order statistic • (Left-leaning) Red-black • Scapegoat • Splay • T • Treap • UB • Weight-balanced	
Heaps	Binary • 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** BST (**D. Sleator** and **R. Tarjan**, **1985**), with no extra information in the nodes (no heights, no colors)
- **Key idea**: after every access, the accessed node is moved to the root using rotations *(splaying)*
- **Motivation**: recently accessed elements become very fast to access again



K goes to the root and the nodes on its path come up too

Splay Trees

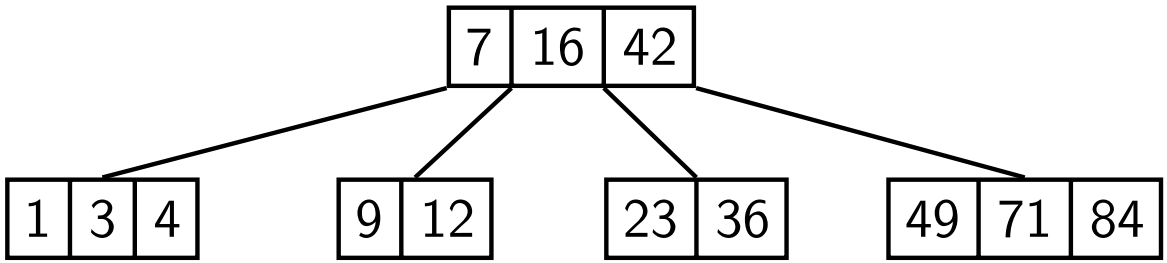
- One operation may take $\Theta(n)$ time (the tree can become a "line"), but splaying a deep node roughly halves the depth of the nodes on its path
- **Guarantee**: any sequence of m operations takes $\mathcal{O}(m \log n)$ time, so each operation costs $\mathcal{O}(\log n)$ **amortized**

  (some operations cost more, others cost less: on average they are $\mathcal{O}(\log n)$)
- Good when some elements are accessed much more often than others (caches, for example); used inside the GCC compiler and the FreeBSD virtual memory system
- If you are curious:
  - Visualization:

    https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
  - Wikipedia and the original paper, "Self-Adjusting Binary Search Trees"

B-Trees

- A **self-balancing search tree** whose nodes have **many keys** and **many children**, introduced by **R. Bayer** and **E. McCreight** in **1970**
- **Motivation**: data on disk, where reading a whole block costs about the same as reading one key: each node fills a block, and paths have few nodes
- A B-Tree of **order** m: each node has at most m children and, except the root, at least $\lceil m/2 \rceil$; **all leaves are at the same depth**



a B-Tree of order 4

- **Insertion**: add the key to a leaf; if the node gets too many keys, it splits in two and the middle key goes up to the parent *(the tree only grows at the root, so it stays perfectly balanced)*

B-Trees in real life

- Real B-Trees have nodes with hundreds of children, so the height is tiny: with 1000 children per node, 3 levels are enough for about a billion keys
- A common variant is the **B+Tree**: data only in the leaves, and each leaf linked to the next one (fast range queries)
- Used for indexing in **filesystems** (Windows NTFS, Linux ext4, MacOS APFS) and in **relational databases** (MySQL, MariaDB, PostgreSQL)













- If you are curious:
  - Visualization:
    `https://www.cs.usfca.edu/~galles/visualization/BTree.html`
  - Wikipedia: B-Tree and B+Tree

Summary

- In a binary search tree, search, insertion, removal, minimum and maximum cost $\mathcal{O}(h)$
- Without balancing, h can be $n-1$ (e.g. sorted insertions); a random order only gives $\mathcal{O}(\log n)$ on average
- **Balanced** binary search trees guarantee $h = \mathcal{O}(\log n)$, using **rotations**:
  - **AVL trees**: the heights of the two subtrees of any node differ by at most 1; $h \lesssim 1.44 \log_2 n$
  - **Red-black trees**: no two consecutive red nodes, same number of black nodes on every path; $h \leq 2 \log_2(n+1)$
- In C++, use `set`, `map`, `multiset` and `multimap` (red-black trees in the common library implementations): $\mathcal{O}(\log n)$ per operation, with the elements always in order