National Research University Higher School of Economics
Faculty of Computer Science
Bachelor's Program "HSE University and University of London Double Degree Program in Data Science and Business Analytics"

Introduction to Programming

Workshop #13

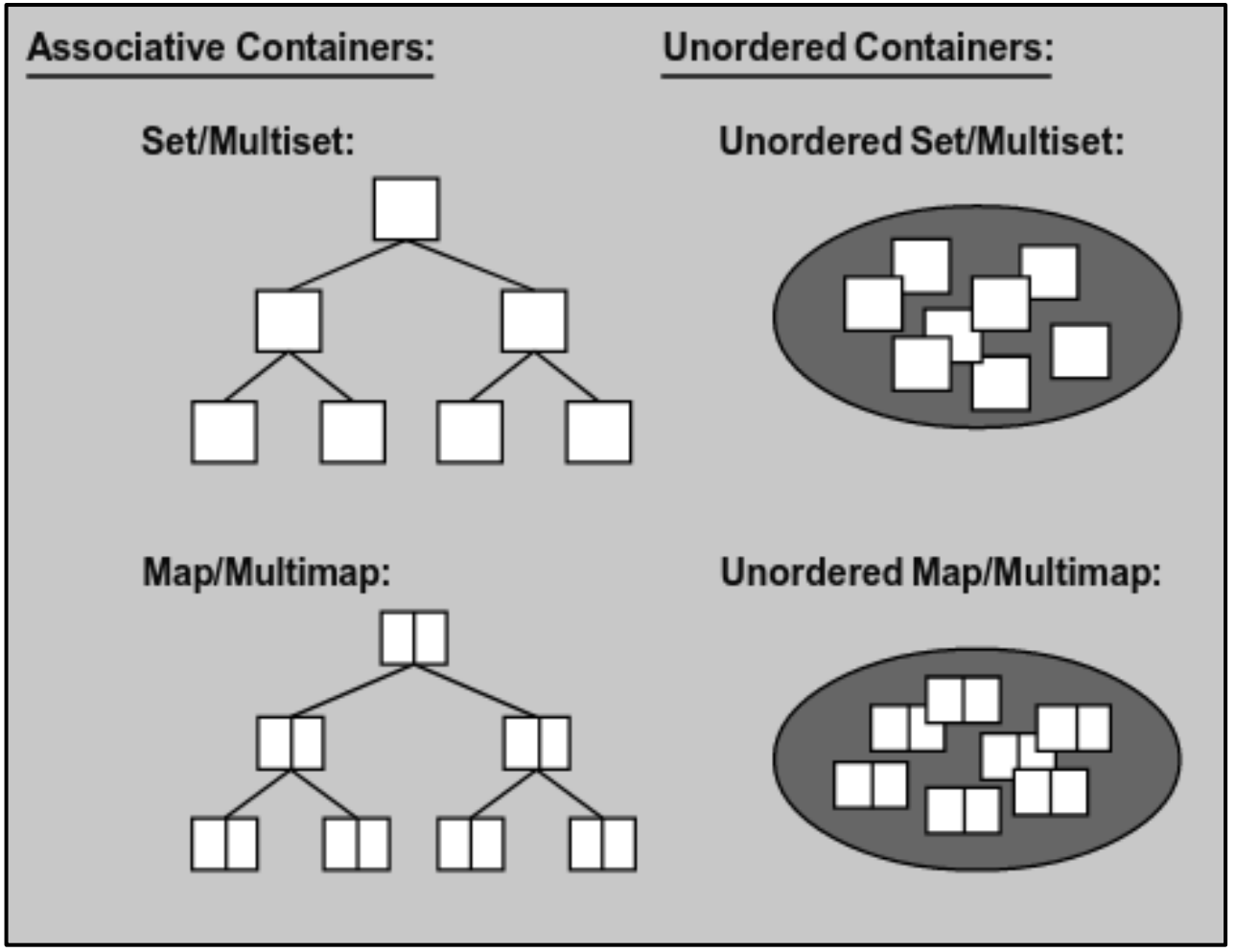
Wed 24.02.2021

Julio Carrasquel

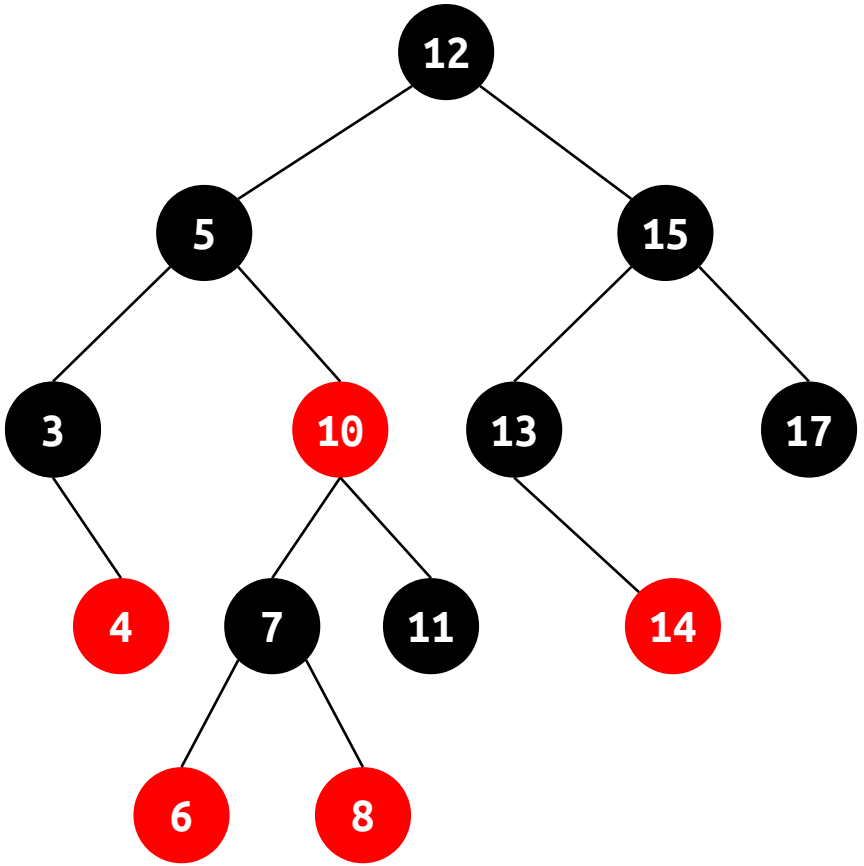
NATIONAL RESEARCH UNIVERSITY

Today's outline

`std:: [unordered_] [multi] maps and sets`



`std:map` and `std:set`- RBTrees

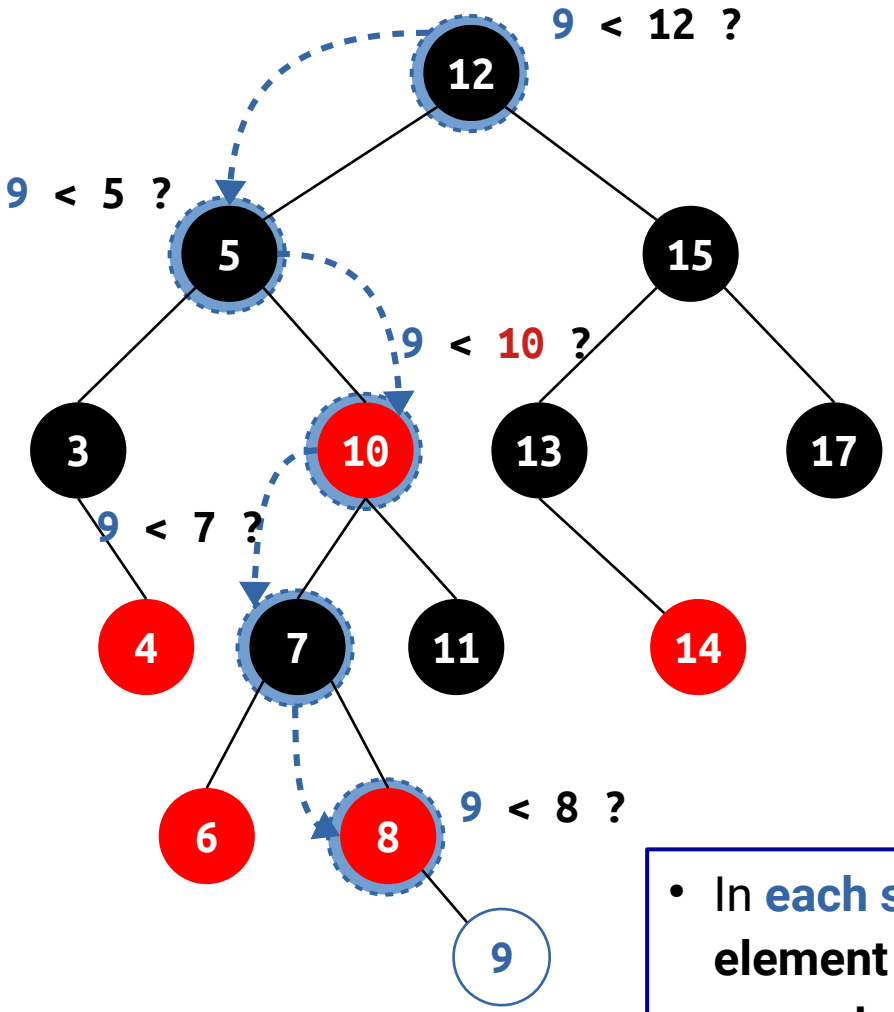


- `std::map` and `std::set` containers are *generally* implemented using Red-Black Trees (RBTrees)
- A RBTree is a *self-balancing* binary search tree.
- A series of rules based on **`red`** and **`black`** colors allow to *maintain balance* in the tree.
- It has *O*(log n) for three basic operations: *`find`*, *`insert`* and *`remove`*.

Play with the example at:
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

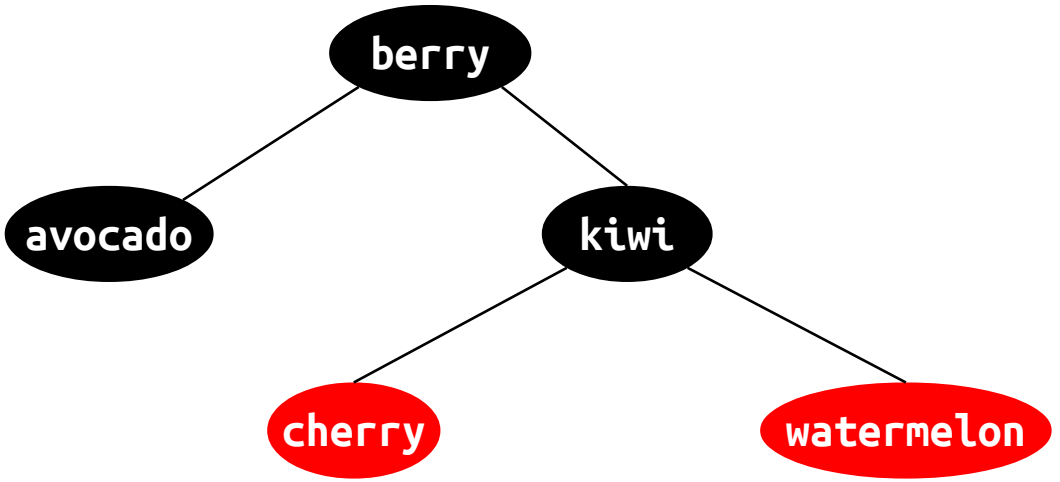
RBTrees

Let's insert element 9



- In **each step**, we check if the **element to insert *is less*** than the **current element.**
- We need operator **<** for that.

RBTrees with `std::string` as key

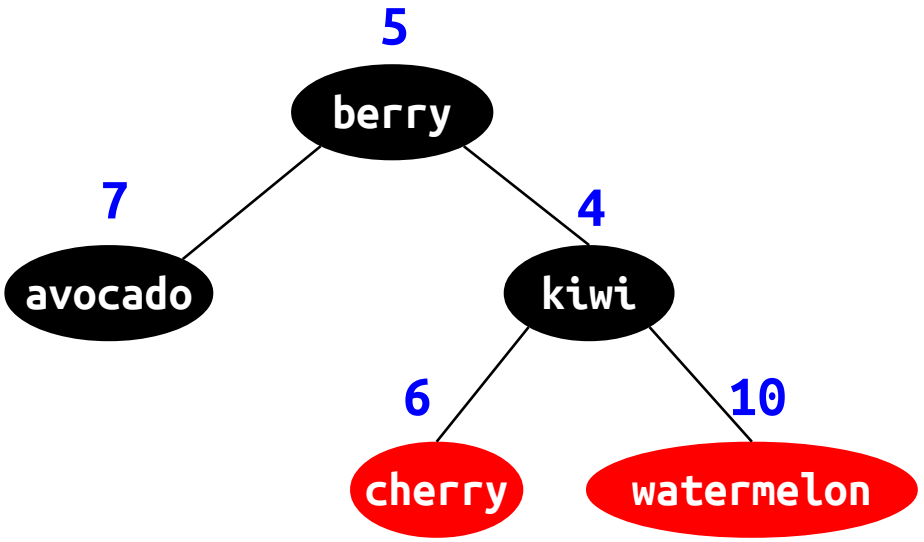


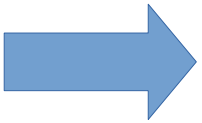
Print order: `avocado berry cherry wiki watermelon`

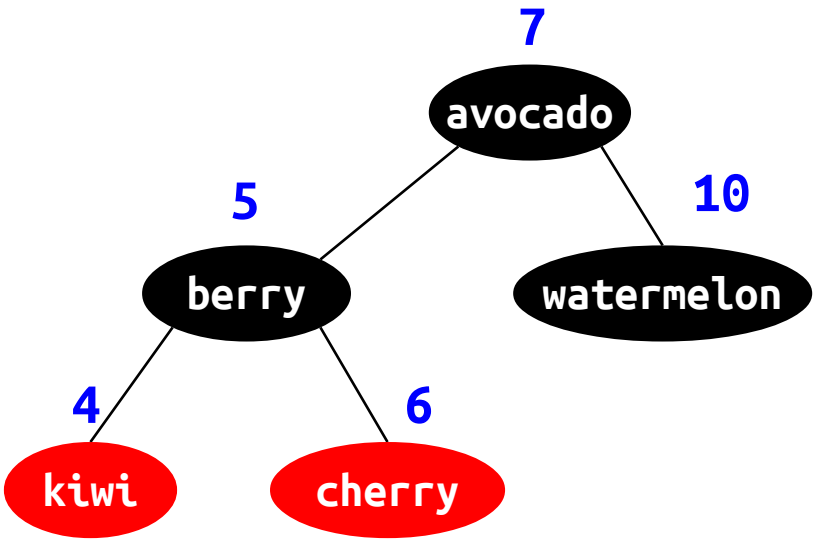
- By default, `std::less<std::string>` sort elements in the containers `std::map` and `std::set` in *lexicographical order.*
- But, we may create a custom comparator for strings.

RBTrees – length of a `std::string` as the key

Create a `std::set<std::string>` where elements are ordered
based on the ***length of a string***
We need to create a custom *compare function*.





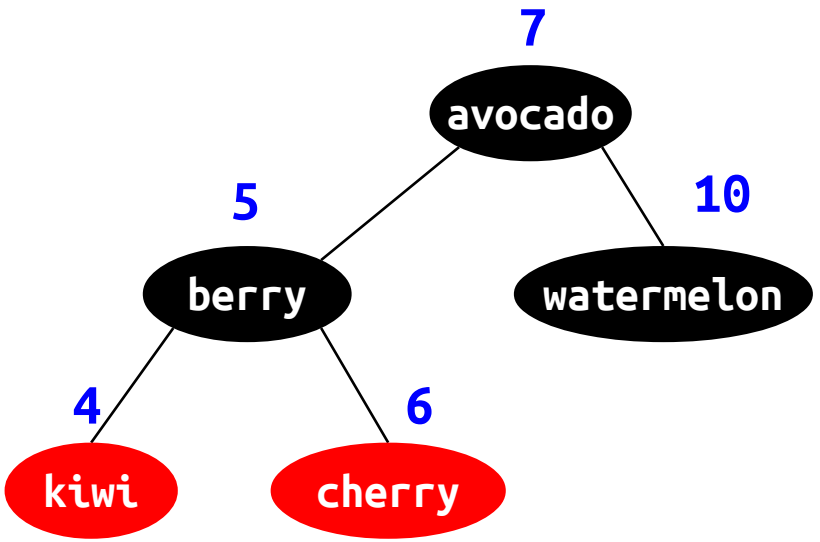


`avocado berry cherry kiwi watermelon`

`kiwi berry cherry avocado watermelon`

RBTrees – length of a `std::string` as the key

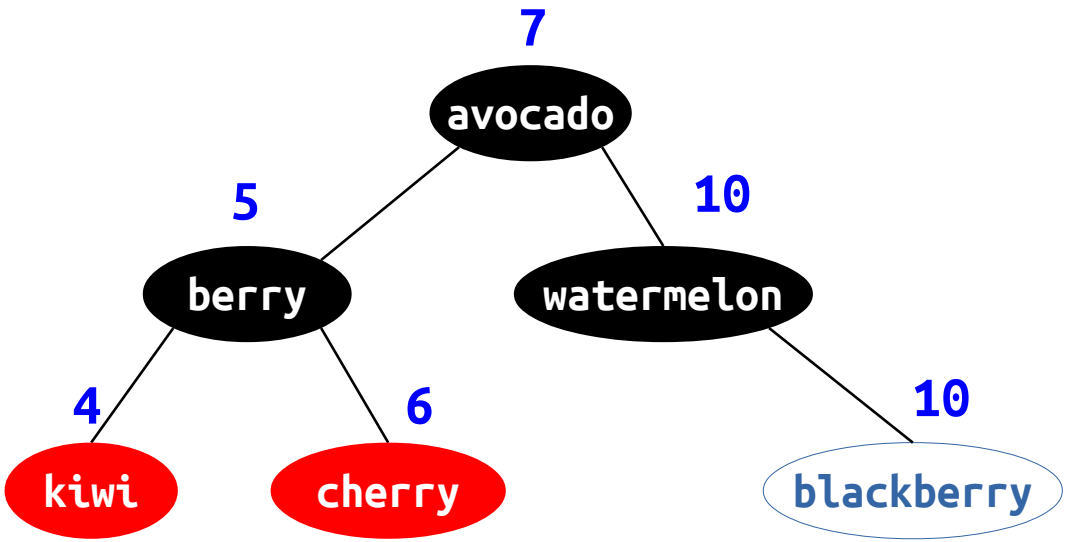
What if we want to add the string "`blackberry`" to the std::set organized by the string length?




```
kiwi berry cherry avocado watermelon
```

RBTrees – length of a `std::string` as the key

What if we want to add the string "`blackberry`" to the std::set organized by the string length? Use `std::multiset`




```
kiwi berry cherry avocado watermelon blackberry
```

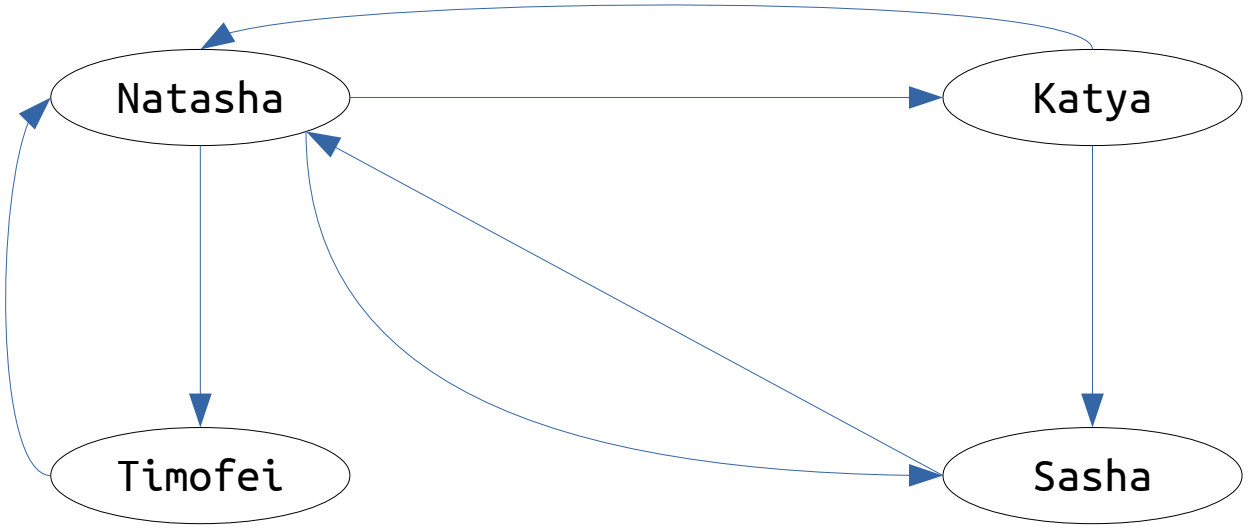
We need to use `std::multiset` to allow multiple keys with the same value!

`std::multiset` (and `std::multimap`) are also implemented with RBTrees, as they still can handle multiple keys with the same value.

`std::multimap` - Example

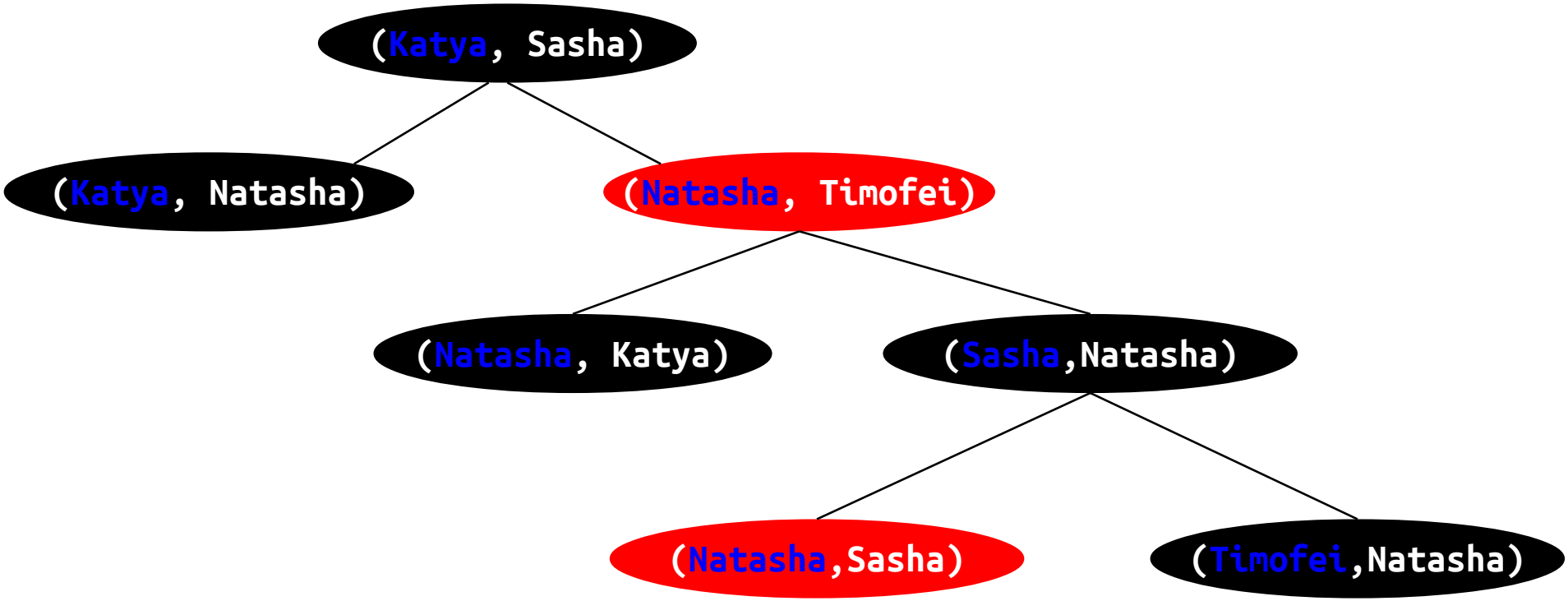
A `std::multimap` can be used to express relationships between elements.

"Who-follows-who" in a social network?



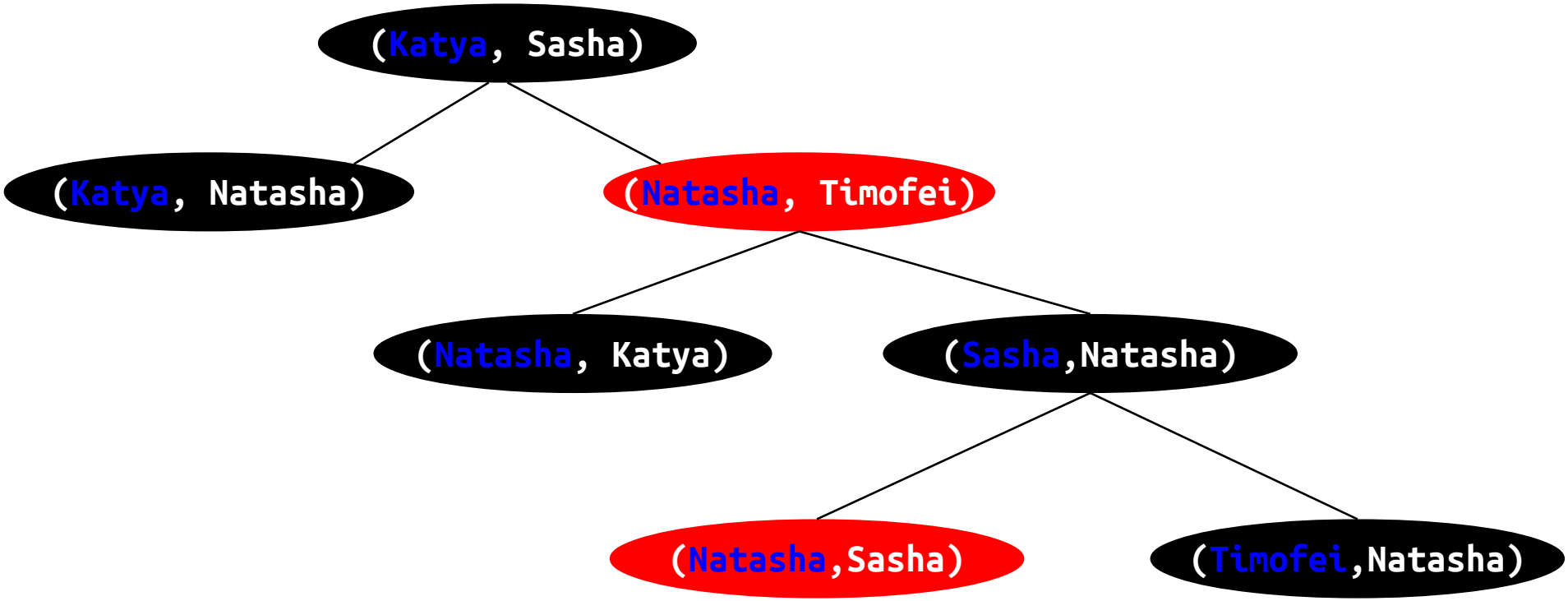
`std::multimap` - Example

Remember that `std::multimap` (and `std::map`) is also generally implemented as a RBTree, where pairs `(key, element)` are ordered by the **key**.



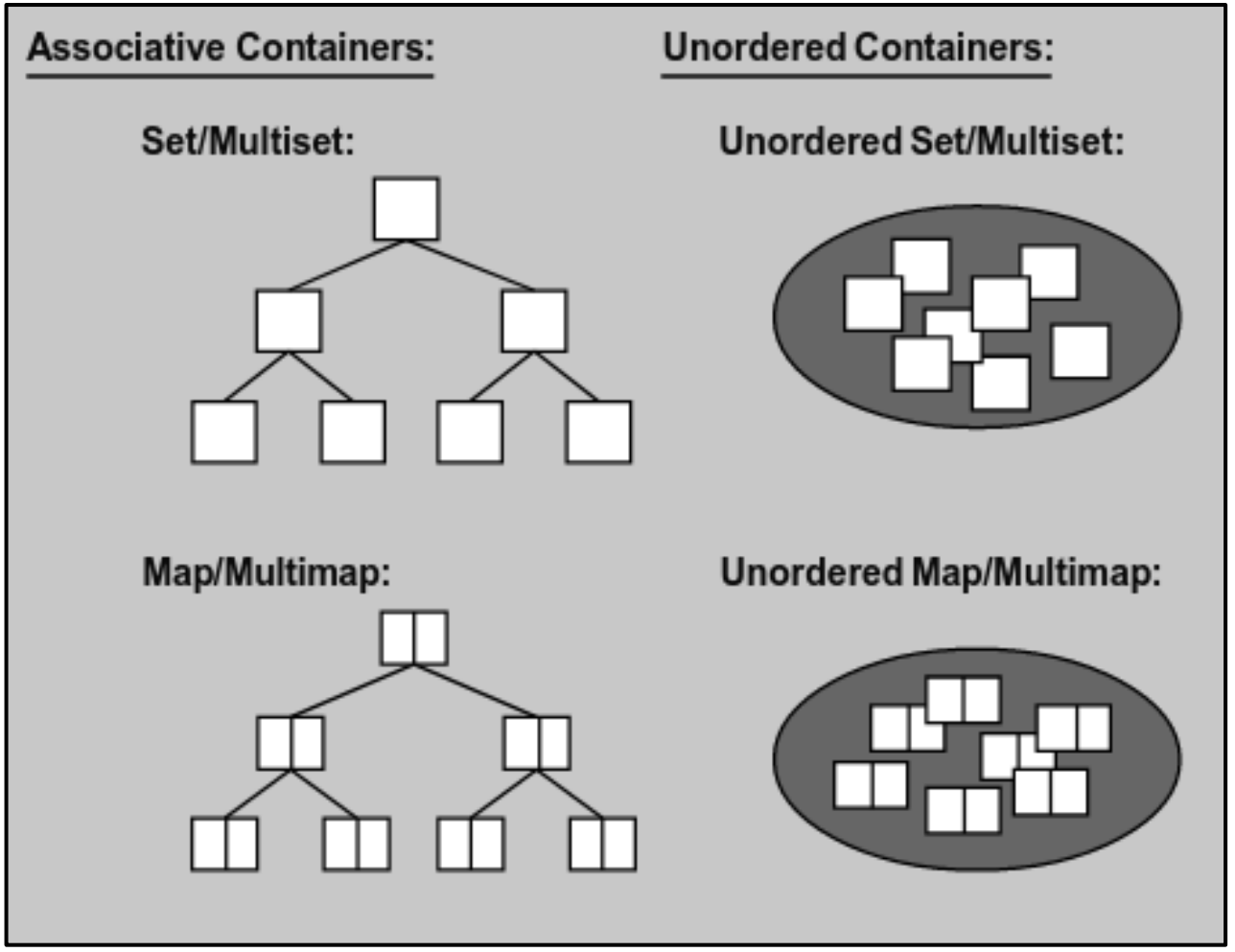
`std::multimap` - Example

Remember that `std::multimap` (and `std::map`) is also generally implemented as a RBTree, where pairs `(key, element)` are ordered by the **key**.



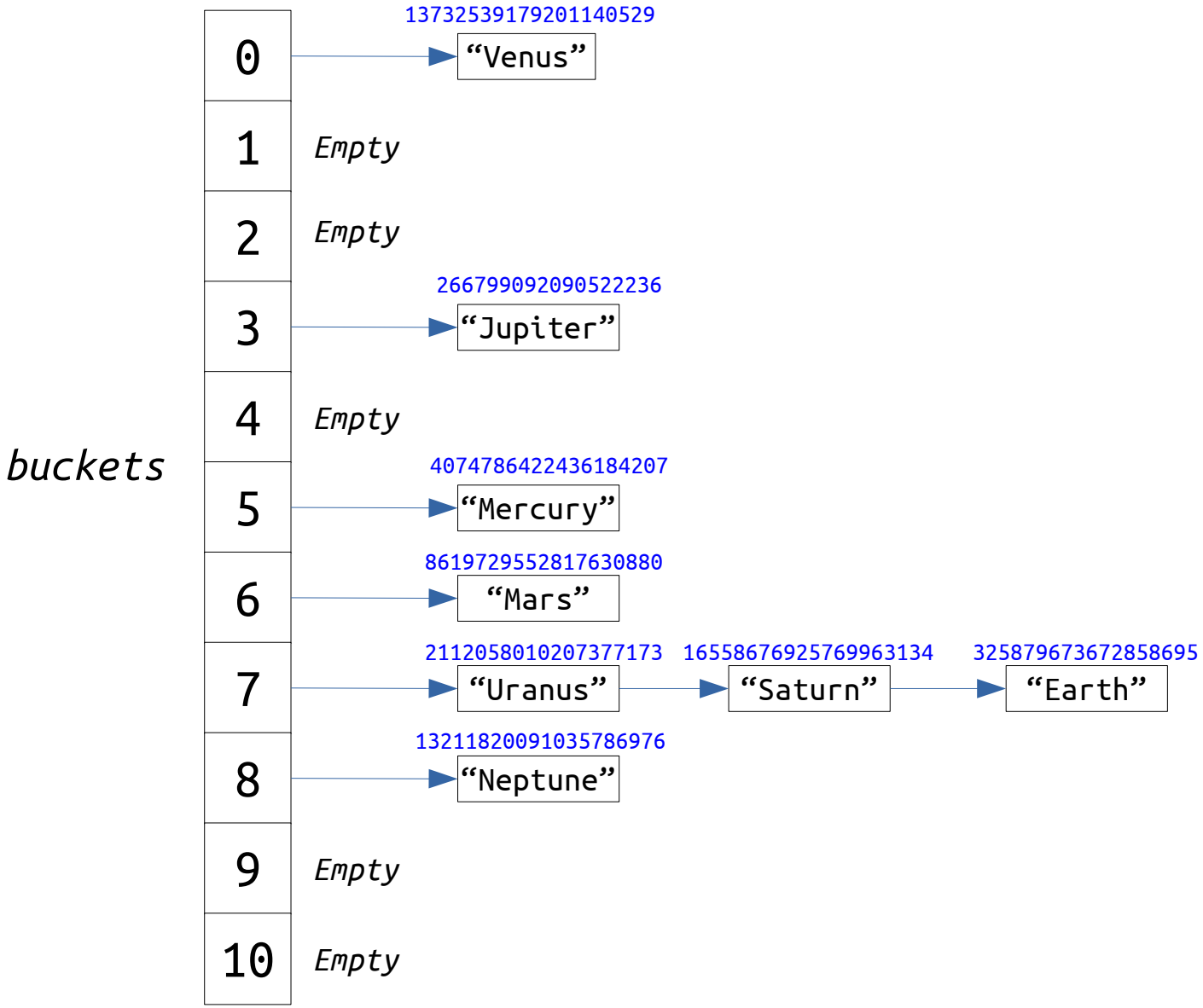
What if it is not be important the internal ordering of elements in a `std::map`, `std::multimap`, `std::set`, or `std::multiset`? Let's use `std::unordered_` !

std:: [unordered_] [multi] maps and sets



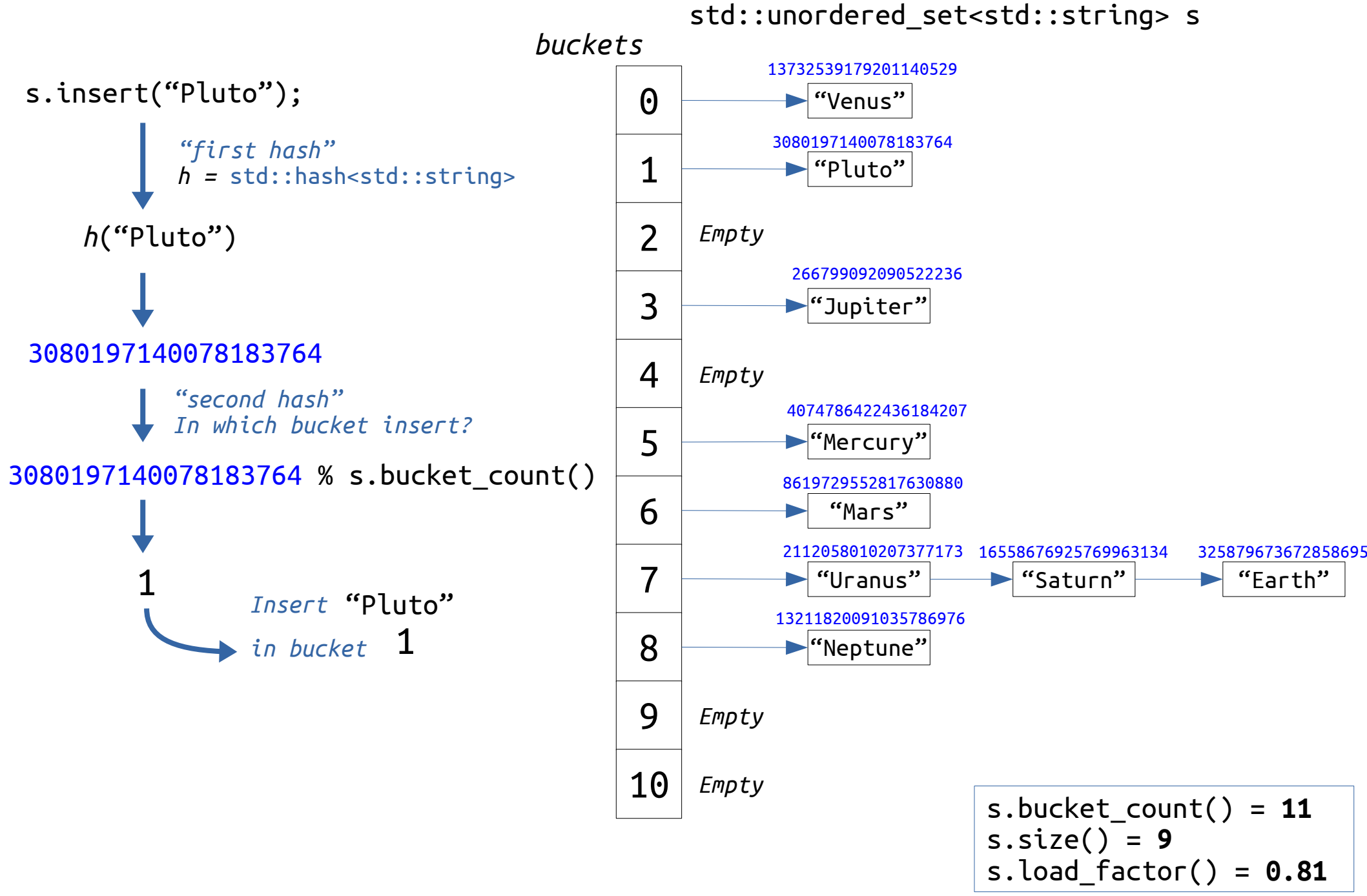
`std::unordered_[multi] set/map` and Hash functions

- `Unordered containers` are *generally* supported by *hash functions*.
- Operations on `unordered containers` (e.g., `find, insert, delete`) are O(1).



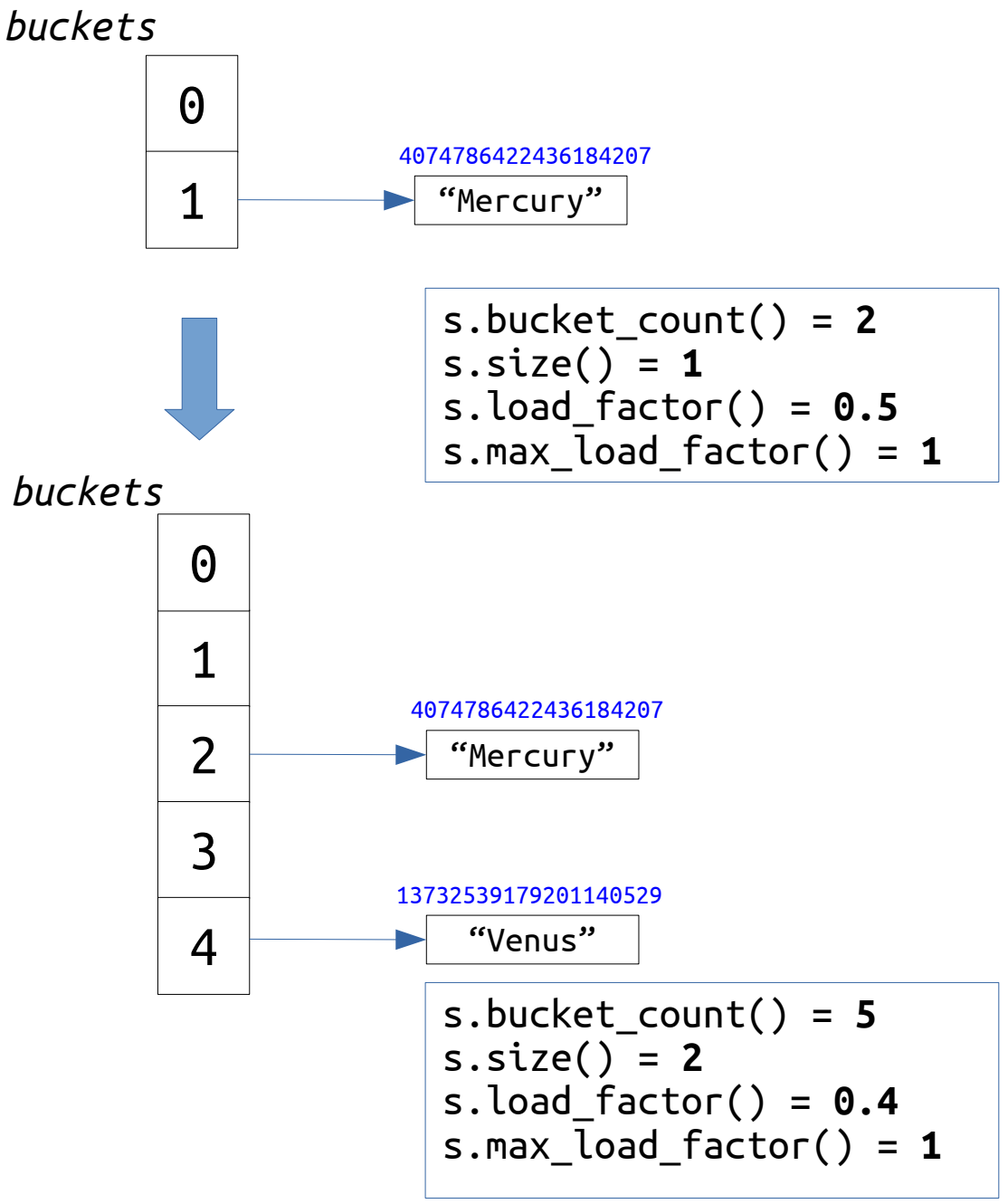
std::unordered_[multi] set/map and Hash functions

std::unordered_[multi] set/map and Hash functions



`std::unordered_[multi] set/map` and Hash functions

`std::unordered_set<std::string> s`



- When inserting a new element, if `load_factor()` > `max_load_factor()` then the table is enlarged (`bucket_count`)
- The *second hash* function "to know in which bucket insert" changes, so elements are re-allocated!

`4076786422436184207 % s.bucket_count()`

The *first hash* `std::hash<std::string>` *does not change!*