Object Oriented Programming with Java

PART 1: JAVA PROGRAMMING BASICS

Collections Part Two: **Advanced usage of collections**, By Aphrodice Rwagaju

Collections

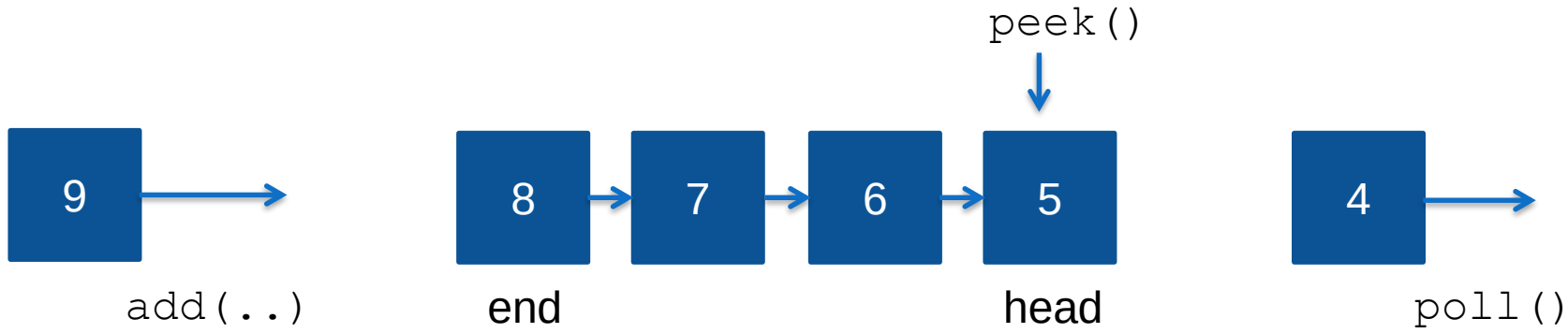
- So far …
  - We've talked about the interfaces:
    - `List`, `Set`, `Map`
  - We've used the classes:
    - `ArrayList`, `HashSet`, `HashMap`

Thread safety

- Threads are different execution units running at the same time (concurrently)
- Some Java classes are not designed to be accessed by multiple threads at the same time, and will give unpredictable results
- Others are designed so and are called thread-safe, e.g.
  - **`Vector`** is a *thread-safe* version of **`ArrayList`**

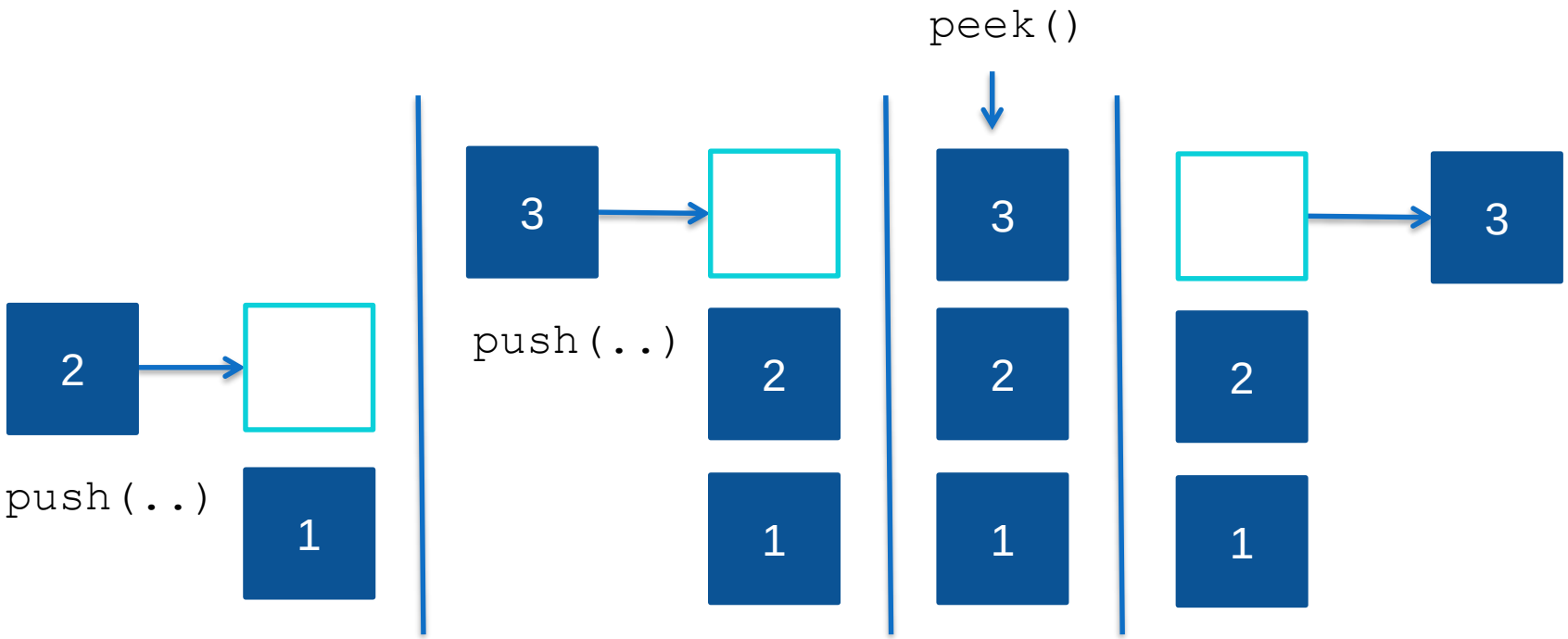
Queues

- These are FIFO (First In First Out) like most queues in real life
- Ideal for storing…
  - Messages to be processed
  - Tasks to completed
- Java has many different queue classes, but the **`LinkedList`** class provides a simple implementation



Stacks

- These are LIFO (Last In First Out)
- Java has a **`Stack`** class which extends **`Vector`**



Equality recap…

- There are two types of equality in Java land…
  - Reference equality (**==**)
  - Object equality (**.equals(…)**)

```
Integer val1 = new Integer(123);
Integer val2 = new Integer(123);
Integer val3 = val1;
```

val1 == val2 **FALSE** val1.equals(val2) **TRUE**

val1 == val3 **TRUE** val1.equals(val3) **TRUE**

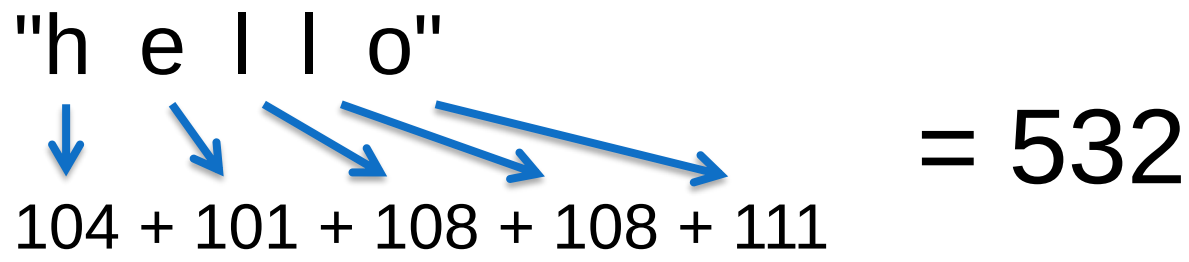
Hash codes

- Some collection classes like **`HashSet`** and **`HashMap`** need to check for duplicates
- Calling **`.equals()`** for every item in the collection could take a long time for complex objects
- To speed things up, Java uses *hash codes*, which are integer representations of an object
- Comparing integers is fast!

Hash functions

- Generally speaking, a hash is a small value (often an integer) generated from a much larger value using a hash function

- Let's define a very simple hashing function for strings which takes the ASCII codes and adds them…



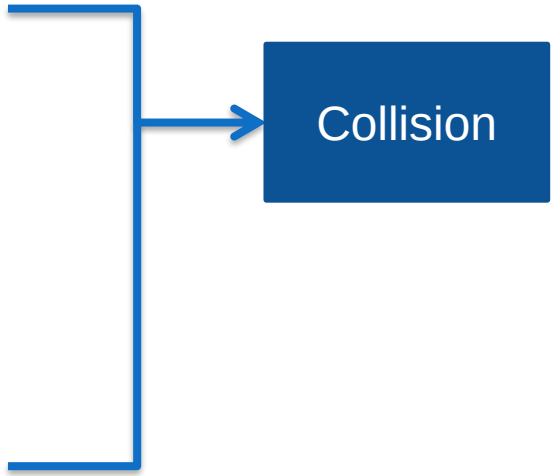
Hashing collisions

- When two inputs generate the same hash output, we call it a collision, e.g.

"h e l l o"
104+101+108+108+111
= 532

"w o r l d"
119+111+114+108+100
= 552

"e h l o l"
101+104+108+111+108
= 532



Hashing uses

- Hashes are also used in cryptography, e.g. hashing passwords
- Good hashing functions:
  - Are quick
  - Minimize collisions
- For cryptographic use they should be:
  - Not too quick
  - Strictly one-way (impossible to calculate the original data)

Hash codes in Java

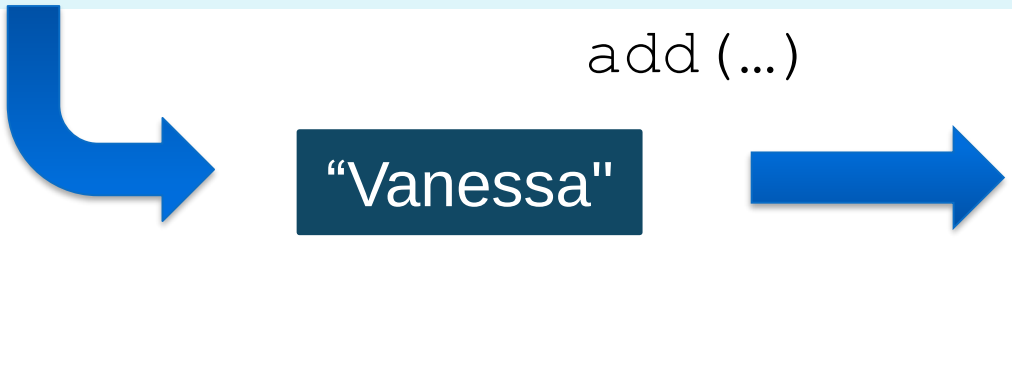
- Every object in Java has a hash code
- These are generated by the **`hashCode()`** method of the **`Object`** class, e.g.

```
String message = "Hello"
int hash = message.hashCode();
```

- The default implementation returns a unique integer based on the memory location of the object

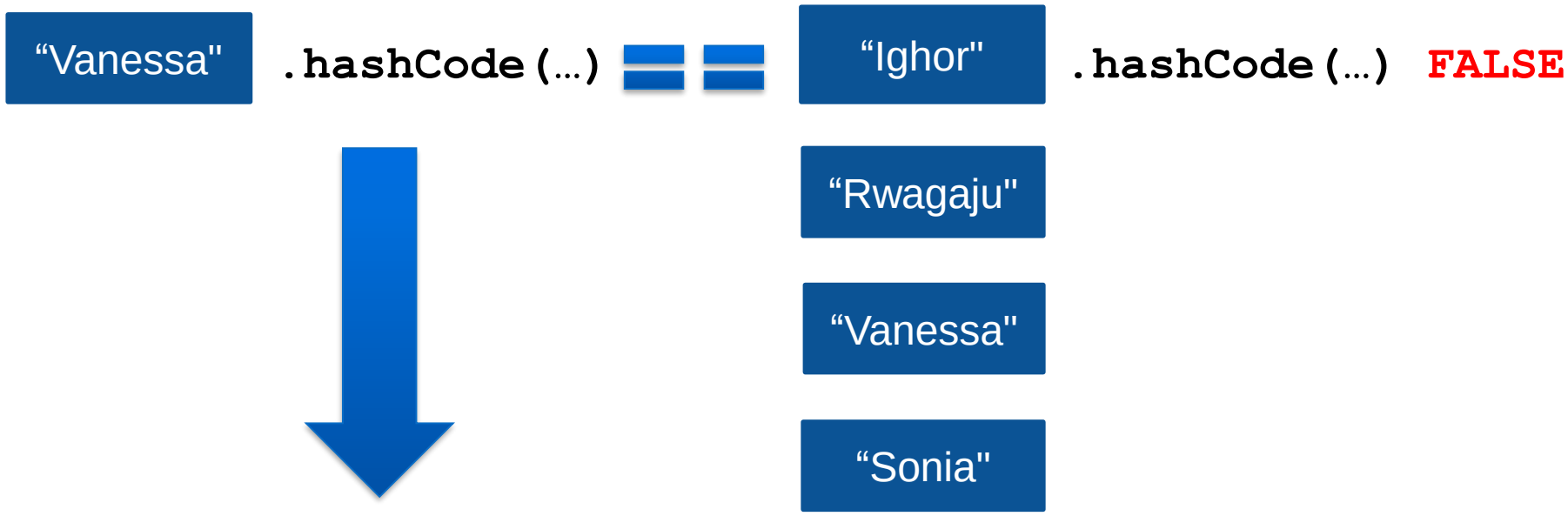
HashSet example

```
HashSet set = new HashSet();
set.add("Ighor");
set.add("Rwagaju");
set.add("Vanessa");
set.add("Sonia");
set.add("Vanessa");
```



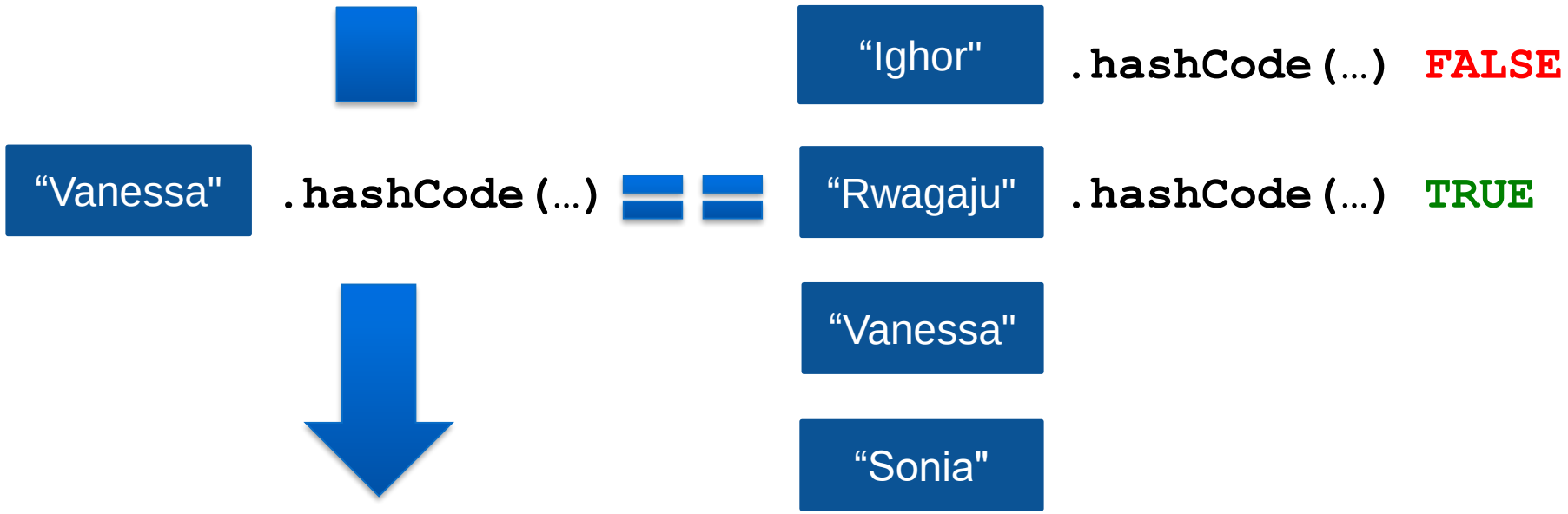
HashSet example

- The hash code of the item being added is compared against each item in the set…
  - Hashcode for Vanessa = 721
  - Hashcode for Rwagaju = 721
  - Hashcode for Ighor = 505
  - Hashcode for Sonia = 506



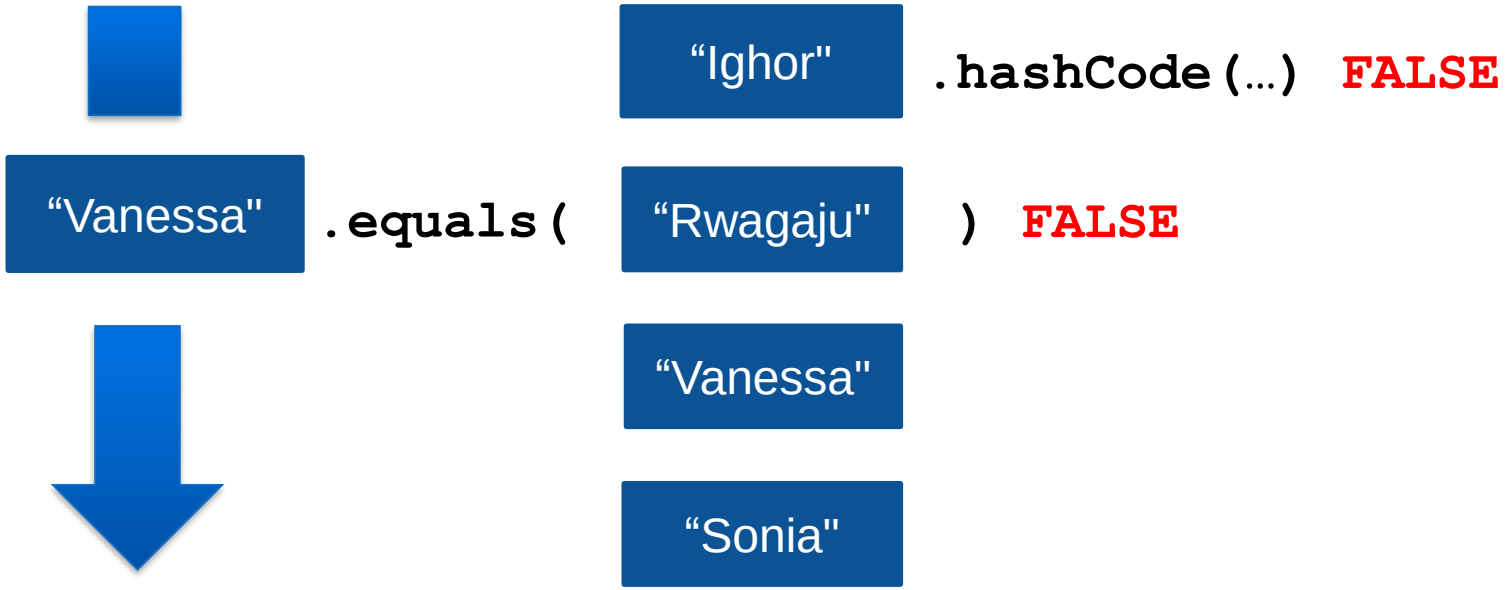
HashSet example

- It's possible that there will be a hash code collision.
  - Hashcode for Vanessa = 721
  - Hashcode for Rwagaju = 721
  - Hashcode for Ighor = 505
  - Hashcode for Sonia = 506



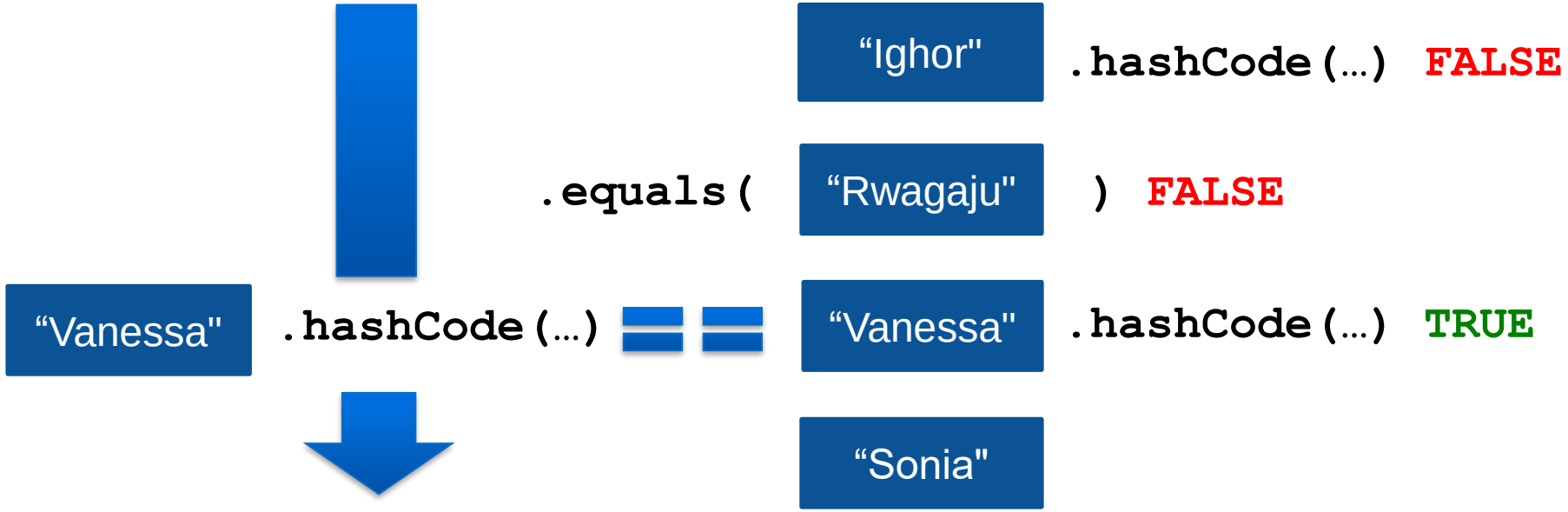
HashSet example

- So if hash codes match, the collection will also check **`equals(…)`**
  - Hashcode for Vanessa = 721
  - Hashcode for Rwagaju = 721
  - Hashcode for Ighor = 505
  - Hashcode for Sonia = 506



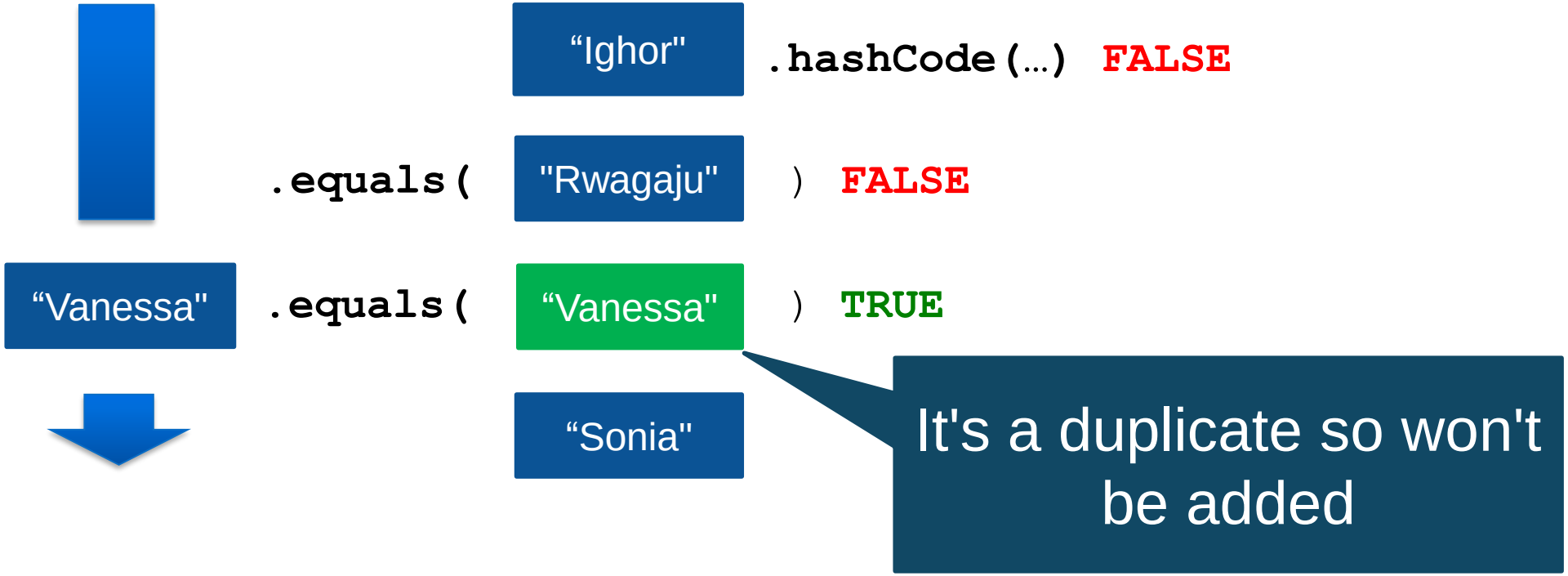
HashSet example

- Hash codes are ***always*** equal when two objects are equal…
  - Hashcode for Vanessa = 721
  - Hashcode for Rwagaju = 721
  - Hashcode for Ighor = 505
  - Hashcode for Sonia = 506



HashSet example

- But we need to check `equals(...)` to be sure
  - Hashcode for Vanessa = 721
  - Hashcode for Rwagaju = 721
  - Hashcode for Ighor = 505
  - Hashcode for Sonia = 506



Hash code rules

- Two objects that are equal (i.e. `.equals(…)` returns true), MUST have the same hash code
- Two objects with the same hash code, are NOT NECESSARILY equal
- Therefore, if you override `.equals(…)` you also need to override `.hashCode()`

Equality example

```
class Book {
	private String title;
	private String author;

	public Book(String title, String author) {
		this.title = title;
		this.author = author;
	}
}
```

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

System.out.println("B1 equals B2: " + b1.equals(b2));
System.out.println("B1 equals B3: " + b1.equals(b3));
```

FALSE

FALSE

Equality example

```
class Book {
        private String title;
        private String author;

        public Book(String title, String author) {
                this.title = title;
                this.author = author;
        }

        public boolean equals(Object obj) {
                Book book = (Book)obj;
                return book.title.equals(this.title) && book.author.equals(this.author);
        }
}
```

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

System.out.println("B1 equals B2: " + b1.equals(b2));
System.out.println("B1 equals B3: " + b1.equals(b3));
```

FALSE

TRUE

Equality example

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

System.out.println("B1 equals B2: " + b1.equals(b2));
System.out.println("B1 equals B3: " + b1.equals(b3));

HashSet set = new HashSet();
set.add(b1);
set.add(b2);
set.add(b3);
System.out.println("Set size: " + set.size());
```

FALSE

TRUE

?

Equality example

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

System.out.println("B1 equals B2: " + b1.equals(b2));
System.out.println("B1 equals B3: " + b1.equals(b3));

HashSet set = new HashSet();
set.add(b1);
set.add(b2);
set.add(b3);
System.out.println("Set size: " + set.size());
```

FALSE

TRUE

3 because the hash codes are different

Equality example

```
class Book {
        private String title;
        private String author;

        public Book(String title, String author) {
                this.title = title;
                this.author = author;
        }

        public boolean equals(Object obj) {
                Book book = (Book)obj;
                return book.title.equals(this.title) && book.author.equals(this.author);
        }

        public int hashCode() {
                return title.hashCode() + author.hashCode();
        }
}
```

If 2 books have same title and author, they are equal…

So if 2 books have same title and author, they should have the same hash code

Equality example

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

System.out.println("B1 equals B2: " + b1.equals(b2));
System.out.println("B1 equals B3: " + b1.equals(b3));

HashSet set = new HashSet();
set.add(b1);
set.add(b2);
set.add(b3);
System.out.println("Set size: " + set.size());
```

2

Generics

- The collection classes support generics
- If you don't use generics you get warnings about type safety
- If you need a collection to hold **any kind of object…**

```
List stuff = new ArrayList();
```

```
List<Object> stuff = new ArrayList<Object>();
```

- And …

```
Set set = new HashSet();
```

```
Set<Object> set = new HashSet<>();
```

```
Map map = new HashMap();
```

```
Map<Object, Object> map = new HashMap<>();
```

Sorting

```
Book b1 = new Book("Hamlet", "Shakespeare");
Book b2 = new Book("The Hobbit", "J.R. Tolkien");
Book b3 = new Book("Hamlet", "Shakespeare");

List<Book> books = new ArrayList<Book>();
books.add(b1);
books.add(b2);
books.add(b3);
Collections.sort(books);
```

The inferred type Book is not a valid substitute for the bounded parameter <T extends Comparable<? super T>>

I.e. we don't know how to sort book objects

Sorting

- To sort a collection, the sorting algorithm needs to know how to compare 2 objects
- We tell it how to compare 2 instances of our class by implementing the `Comparable` interface and it's `compareTo` method
- `String` class implements `Comparable` and `compareTo` to sort string alphabetically

Implementing compareTo

```
public int compareTo(Book book)
```

Current book (`this`) is object 1

Object 2

Condition	Returns
Object 1 < object 2	< 0
Object 1 equals object 2	0
Object 1 > object 2	> 0

Implementing compareTo

- How we implement **`compareTo`**, determines how the items will be sorted, e.g.
  - To sort by title A-Z…

```
public int compareTo(Book book) {
    return this.title.compareTo(book.title);
}
```

  - To sort by author A-Z…

```
public int compareTo(Book book) {
    return this.author.compareTo(book.author);
}
```

Implementing compareTo

- To sort by title Z-A…

```
public int compareTo(Book book) {
    return -this.title.compareTo(book.title);
}
```

- or…

```
public int compareTo(Book book) {
    return book.title.compareTo(this.title);
}
```

Implementing compareTo

- The `String compareTo` method compares Unicode character values – its *case sensitive*

- I.e. A < B < … < Z < a < b < … < z
  - To sort by title A-Z case insensitive…

```
public int compareTo(Book book) {
    return this.title.toLowerCase()
              .compareTo(book.title.toLowerCase());
}
```

Implementing compareTo

- Sorting on multiple fields
  - To sort by title, then author…

```
public int compareTo(Book book) {
      // Sort by title first
      int byTitle = this.title.compareTo(book.title);
      if (byTitle != 0)
            return byTitle;

      // If equal, then sort by author
      return this.author.compareTo(book.author);
}
```

Comparator

- `Comparable (compareTo)` requires than you can modify the class that you are trying to sort
- The `Comparator` class allows us to define a compare method for classes we can't modify

Comparator Example

- Supposing we want to sort strings by length rather than A-Z
- We can't modify **`String.compareTo`**, but we can create a new **`Comparator...`**

```
class StringLengthComparator implements Comparator<String> {
     public int compare(String str1, String str2) {
          return str1.length() - str2.length();
     }
}
```

Comparator Example

- To use the comparator, we pass an instance of it to the **`sort`** method…

```
List<String> names = new ArrayList<String>();
names.add("Ethan");
names.add("Jazzy");
names.add("Dorcas");

Collections.sort(names, new StringLengthComparator());
```

Trees

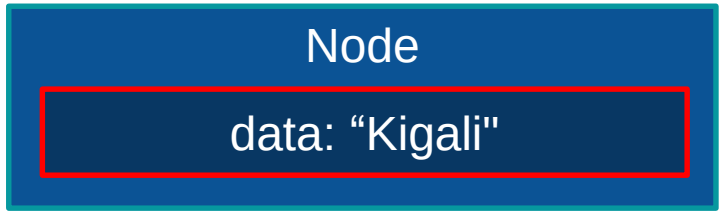
- Sorting a collection can be time-consuming
- Tree's are a type of collection where the order of items is always maintained
- When a new item is added, it is added "in order"
- A commonly used Java tree class is **`TreeSet`**

```
TreeSet<String> cities = new TreeSet<String>();
```

Trees

- Items in a tree are attached to nodes
- We add an item to our tree, and it comes a node…

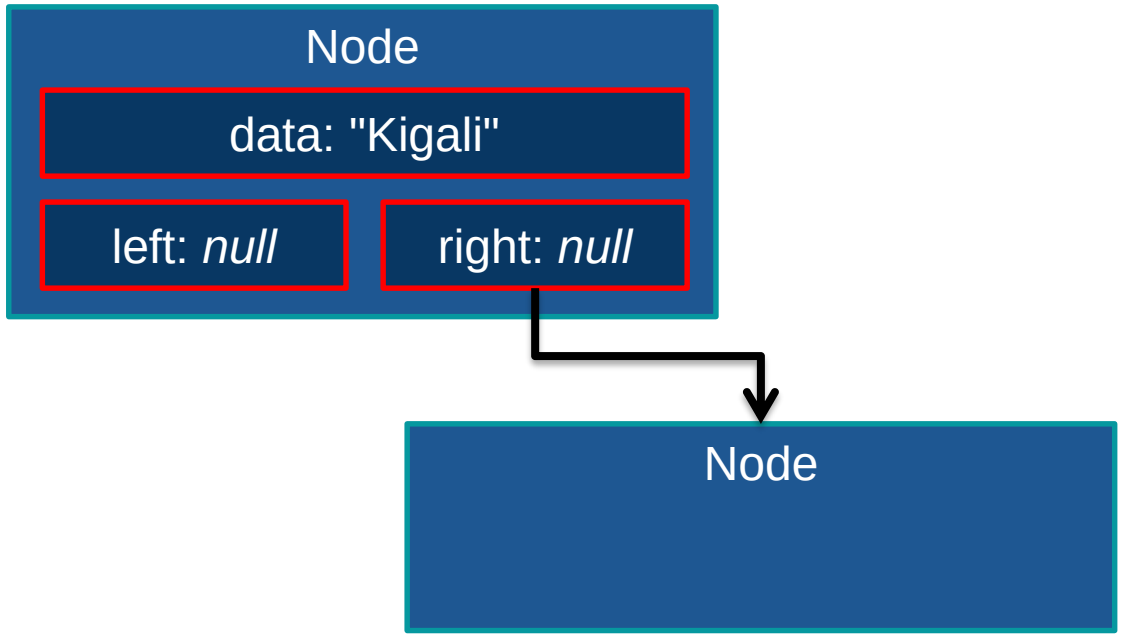
```
TreeSet<String> cities = new TreeSet<String>();
cities.add("Kigali");
```



- The first item becomes the ***root*** node

Trees

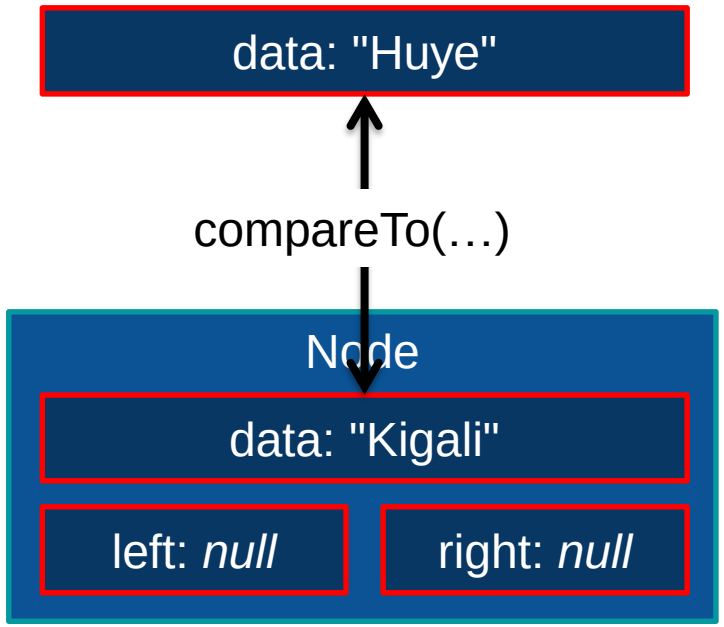
- Every node has two references to allow other nodes to be connected to it



Trees

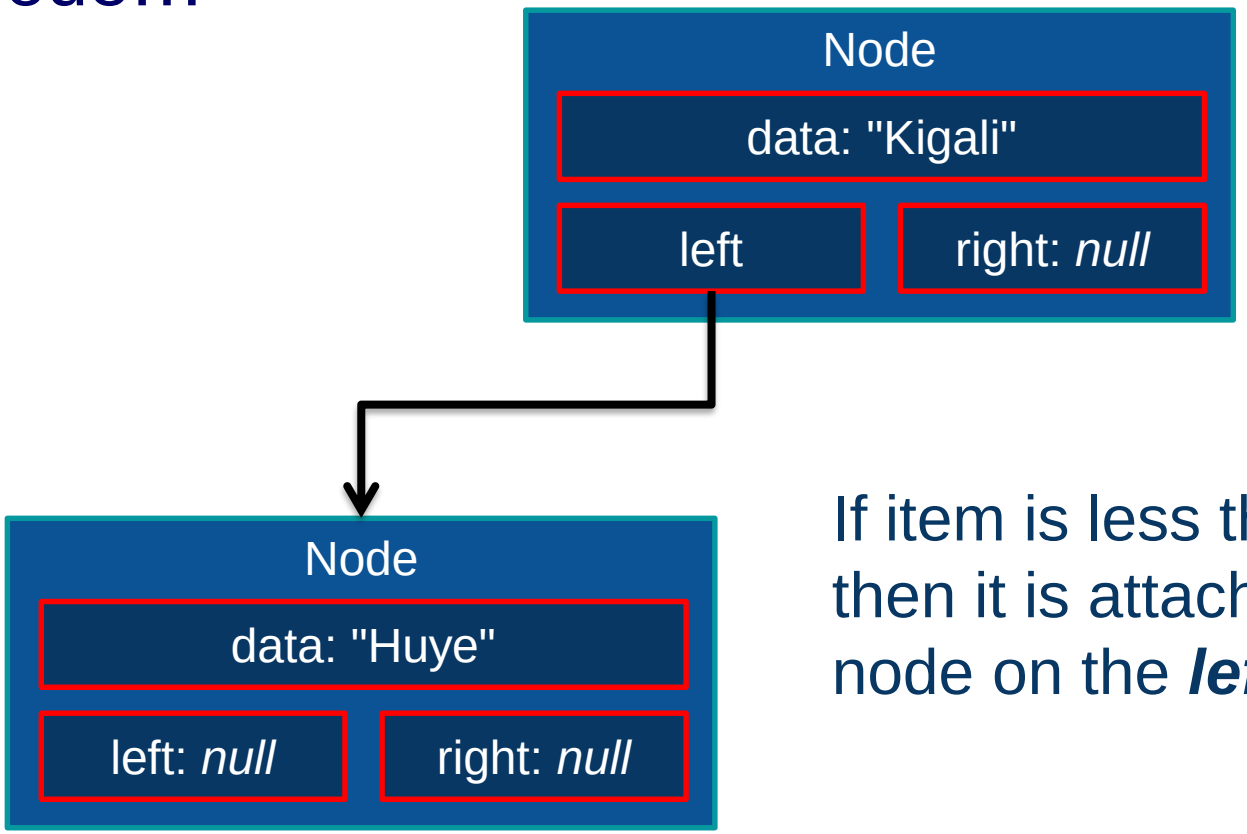
- When a new item is being inserted, it is compared with the root node…

```
cities.add("Huye");
```



Trees

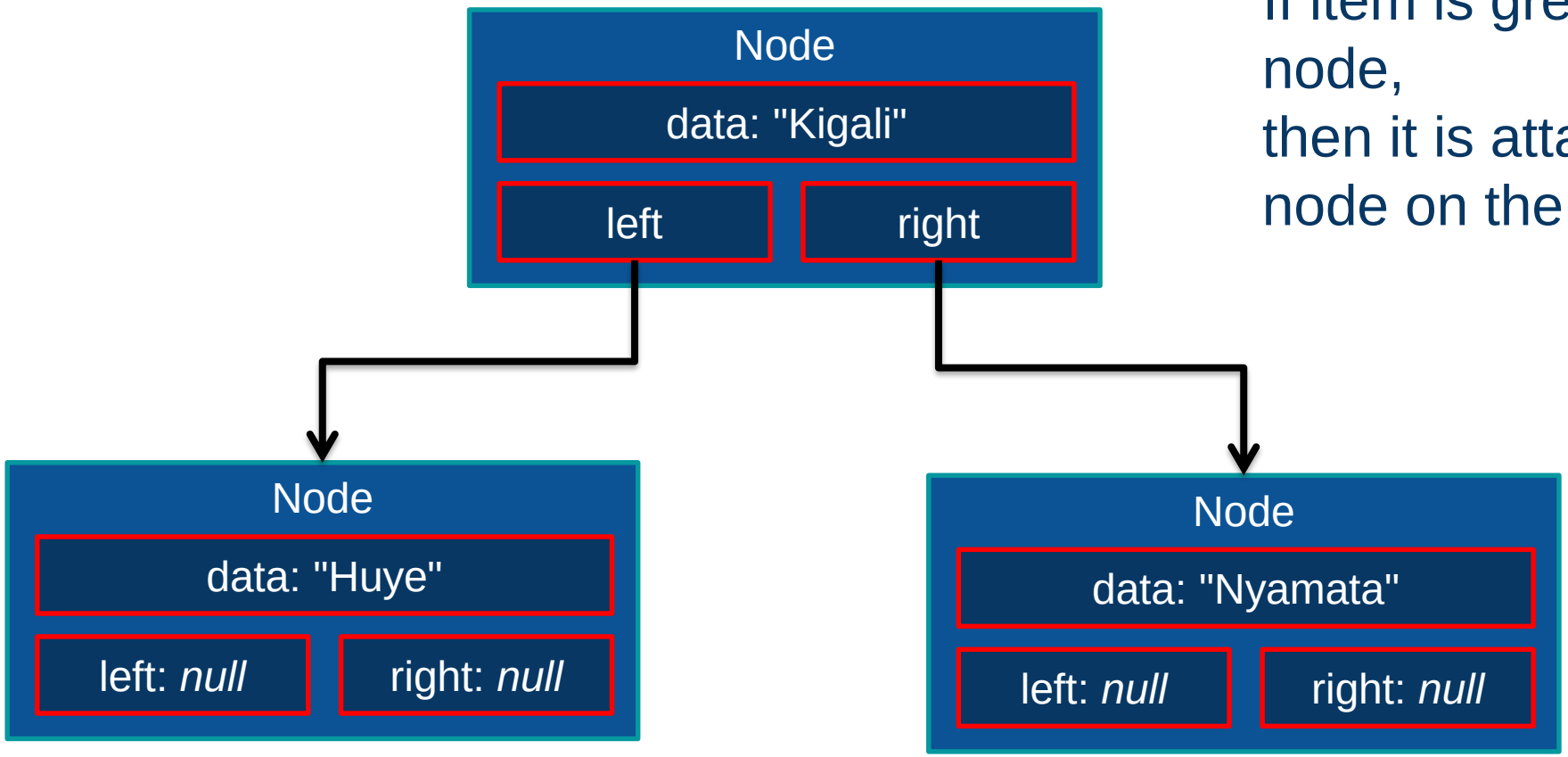
- When a new item is being inserted, it is compared with the root node…



If item is less than the node, then it is attached as a new node on the ***left*** reference

Trees

```
cities.add("Nyamata");
```

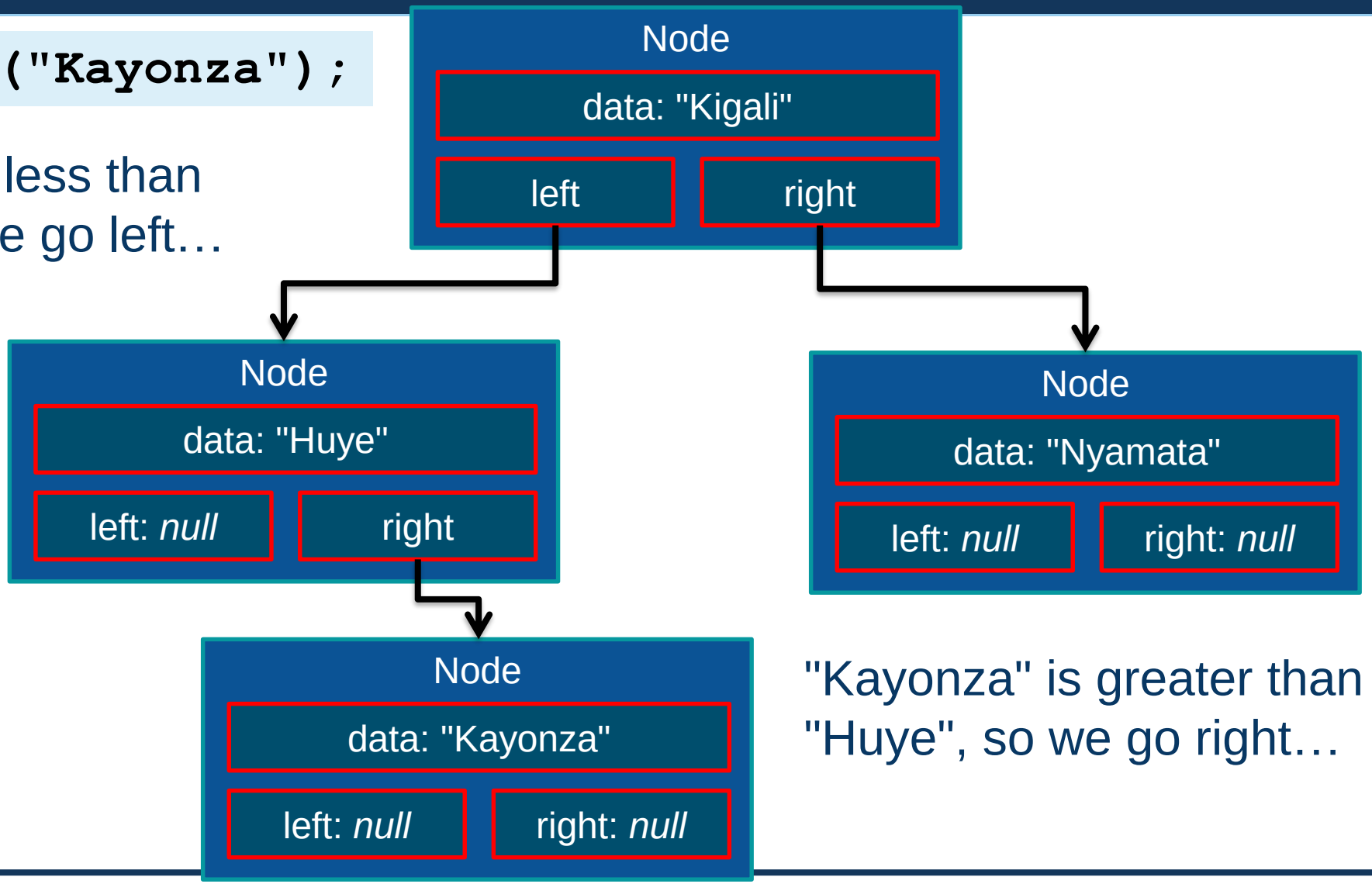


If item is greater than the node,
then it is attached as a new node on the ***right*** reference

Trees

```
cities.add("Kayonza");
```

"Kayonza" is less than "Kigali", so we go left…



"Kayonza" is greater than "Huye", so we go right…

Trees

- Trees require more time to insert items
- But we can access items in order very efficiently using a recursive algorithm
- This called "***walking***" the tree

Trees

- In pseudo-code such a walking algorithm looks something like…

```
walk (node)
    if node.left not null
        walk (node.left)

    process node.data

    if node.right not null
        walk (node.right)
end
```

Calls itself (recursive)

Do something to each item of data in order

- We start it on the root node, i.e. **`walk(root)`**

Reference

- https://docs.oracle.com/javase/tutorial/collections/index.html

EoF