RationalHashingLab

Generated by Doxygen 1.8.6

Wed Nov 25 2015 12:31:55

Contents

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 Class Documentation

3.1 HashTbl< E, V > Class Reference

Classes

- class **Entry**

Public Member Functions

- HashTbl ()
- void put (E key, V value)
- V get (E key)
- Iterator< E > keys ()
- Iterator< V > values ()

3.1.1 Detailed Description

This is a truly minimal implementation of the well-known HashTable class that is still defined in Java (qv). Essentially, a HashTable allows users to associate values with keys in O(1) time (amortized over the life of the running application).

Note: this implementation throws NullPointerExceptions if `put` is called with either a null key or a null value.

Moreover, instead of returning Enumerations (old school), this version returns Iterators for its keys and values.

A note on Iterators: The iterators that we are using here are *not* true `Iterator`s in the following sense: they are not *thread-safe*. Unlike Java's standard `Iterator` implementations, our iterators are just copies of the current state that are created as needed and given to the client. Real iterators, on the other hand, are more directly connected with the underlying collection, meaning that a real iterator needs to `throw` a `ConcurrentModification-Exception`. In the literature, objects that are sensitive to changes made outside of their purview are called *fail-fast*. Naturally, removing an entry through the Iterator's `remove` method is acceptable, but any other modification should raise the concurrent modification exception—more about this in class.

You could theoretically use objects of this class as a hash table, but too much would still need to be done for real applications. Curious students should see the documentation for the `HashTable` class in the online API (from Oracle).

Author

UMD CS Department.

Parameters

<*E*>	///> Keys type
<*V*>	///> Values type.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 HashTbl ( )

Constructs a Hash Table containing a default number of empty buckets.

Note: a "production strength" version of this class would likely provide additional constructors that allowed us to control growth rate, initial size, etc.

3.1.3 Member Function Documentation

3.1.3.1 V get (E *key*)

Returns the value associated with `key`. Because this is a table, nulls are not allowed, therefore if a `null` is returned ... we know that the key wasn't found.

Parameters

key	

Returns

3.1.3.2 Iterator<E> keys ( )

Returns an Iterator over the `key`s in this table; note, no particular order is specified here.

Returns

an Iterator over Keys.

3.1.3.3 void put (E *key*, V *value*)

Installs the `value` on the `key` in this table. Note, if either parameter is `null` a `NullPointerException` is signaled.

Parameters

key	
value	

3.1.3.4 Iterator<V> values ( )

Returns an Iterator over the `value`s in the table; note, no particular order is assumed.

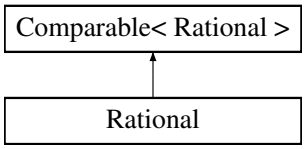
Returns

The documentation for this class was generated from the following file:

- HashTbl.java

3.2 Rational Class Reference

Inheritance diagram for Rational:



Public Member Functions

- Rational (int num, int denom)
- Rational (Rational rational)
- int getNumerator ()
- int getDenominator ()
- Rational plus (Rational other)
- Rational diff (Rational other)

- Rational mult (Rational other)
- Rational divide (Rational other)
- String toString ()
- boolean equals (Object other)
- int hashCode ()
- int compareTo (Rational other)

Protected Member Functions

- Rational recipricol ()

3.2.1 Detailed Description

A streamlined (simplified) implementation of a class that represents non-negative rational numbers. Recall that a rational number is an object that contains two integers, call them a and b, such that a/b and b not equal 0.

This implementation needs to support comparability as well as the basic four arithmetic operators, +,-,∗ and /.

In addition, the `Rational` class must override the `equals`, the `toString()`, and the `hashCode` methods.

Author

UMD CS Dept.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Rational (int *num*, int *denom*)

The main constructor for objects of this type. A rational object contains two integers, one for its numerator and another for its denominator, which cannot be zero. Note: throws an `IllegalArgumentException` in the event that the `denom` argument is zero.

Parameters

num	
denom	

3.2.2.2 Rational (Rational *rational*)

Standard copy-ctor implementation.

Parameters

rational	

3.2.3 Member Function Documentation

3.2.3.1 int compareTo (Rational *other*)

Implements the Comparable interface for Rational number objects.

3.2.3.2 Rational diff (Rational *other*)

Creates a new `Rational` object that embodies the "difference" of `this` and the `other` rational.

Important Note

Because this implementation only provides non-negative rational numbers, if the difference between this Rational and the other Rational number is less than zero, then a Rational number whose numerator is zero and whose denominator is non-zero is returned; this, in effect, is a Rational number representing zero.

Parameters

other	

Returns

3.2.3.3 Rational divide (Rational *other*)

Creates a new `Rational` object embodies the "quotient" of `this` and the `other` rational.

Important Note:

This method should throw an `ArithmeticException` informing the user that division by zero has been attempted in the event that either operand is zero.

Parameters

other	

Returns

3.2.3.4 boolean equals (Object *other*)

[Override, required] Returns true only when `this` rational number equals the `other` by the rules of algebra. For example: 1/2 equals 2/4, or any other rational that obeys the algebraic relationship.

Important Consideration

Because the Rational class supports only non-negative rational numbers, great care should be exercised to ensure that any two Rational objects that are zero, e.g., 0/1 and 0/2, are equal because these are, in effect, zero.

3.2.3.5 int getDenominator ( )

Returns the denominator of this `Rational` object.

Returns

3.2.3.6 int getNumerator ( )

Returns the numerator of this `Rational` object.

Returns

3.2.3.7 int hashCode ( )

[Required] Overrides the default `hashCode` method to ensure the equals contract for hashes, i.e.,two `Rational` objects that are `equal` must compute the same hash code.

Important Consideration

Because the Rational class supports only non-negative rational numbers, the reference of any Rational whose value is zero (i.e., whose numerator is zero), must hash to an object that represents a unique Rational that represents zero.

3.2.3.8 Rational mult (Rational *other*)

Creates a new `Rational` object that embodies the "product" of `this` and the `other` rational.

Parameters

other	

Returns

3.2.3.9 Rational plus (Rational *other*)

Creates a new `Rational` object that embodies the "sum" of `this` and the `other` rational.

Parameters

other	

Returns

3.2.3.10 Rational recipricol ( ) [protected]

[Optional method] Returns the multiplicative inverse of this rational.

Returns

3.2.3.11 String toString ( )

Note: this method must render the `Rational` exactly as `numerator/denominator`.

The documentation for this class was generated from the following file:

- Rational.java

Index