Kathryn Hodge

203 Software Development

November 7, 2014

API for Polymorphic Finite Set Bags

Interface Bag<D extends java.lang.Comparable>

All Superinterfaces: Sequenced<D>

All Known Implementing Classes: SetBag_Empty, SetBag_NonEmpty

public interface Bag<D extends java.lang.Comparable> extends Sequenced<D>

Description: A bag is a multi-set, which can hold duplicates, containing any kind of data.

Method Summary: returnValue methodName(dataType Param1, dataType Param2)

```
int cardinality();
int getCount(D elt);
boolean isEmptyHuh();
boolean member(D elt);
Bag remove (D elt);
Bag removeN(D elt, int n);
Bag removeAll(D elt);
Bag add(D elt);
Bag addN(D elt, int n);
Bag union(Bag u);
Bag inter(Bag u);
Bag diff(Bag u);
boolean equal (Bag u);
boolean subset (Bag u);
String toStringBST();
Sequence<D> seq();
int sumIt ();
int sumItS(Sequence<D> as);
String stringIt();
String stringItS(Sequence<D> as);
Bag<D> addInner(D elt, int n);
public boolean isRedHuh();
```

Method Detail:

Note: All of these methods must be called on Bag instances.
Note: D is a generic-type
Note: In the examples, curly braces {} represent Bag data structures and brackets [ ] represent sequence data structure.

cardinality:

Signature: public int cardinality();
Params: None
Returns: int
Description: This method returns the size of the Bag.
Example: {1, 2, 3, 3}.cardinality() = 4

getCount;

Signature: public int getCount(D elt);
Params: D elt
Returns: int
Description: This method returns the number of times the element elt is in the Bag.
Example: {1, 2, 3, 3}.getCount(3) = 2

isEmptyHuh:

Signature: public boolean isEmptyHuh();
Params: None
Returns: boolean
Description: This method return true if this Bag is empty, false if it is not empty
Example: {1, 2, 3, 3}.isEmptyHuh() = false; {}.isEmptyHuh() = true;

member:

Signature: public boolean member(D elt);
Params: D elt
Returns: boolean
Description: This method return true if the element is a member of the set; if the elt is not a member of the set, then it returns false.
Example: {1, 2, 3, 3}.member(1) = true; {1, 2, 3, 3}.member(4) = false

remove:

Signature: public Bag remove (D elt);
Params: D elt

Returns: Bag
Description: This methods returns a new bag, taking away one instance of the elt
Example: {1, 2, 3, 3}.remove(3) = {1, 2, 3}

<u>removeN:</u>
Signature: public Bag removeN(D elt, int n);
Params: D elt, int n
Returns: Bag
Description: This method returns a new bag with n less instances of elt
Example: {1, 2, 3, 3}.remove(3, 2) = {1, 2}

<u>removeAll:</u>
Signature: public Bag removeAll(D elt);
Params: D elt
Returns: Bag
Description: This method returns a new bag without elt
Example: {1, 2, 3, 3, 4, 4, 4, 4}.removeAll(4) = {1, 2, 3, 3}

<u>add:</u>
Signature: public Bag add(D elt);
Params: D elt
Returns: Bag
Description: This method returns a new bag with one added instance of elt
Example: {1, 2}.add(2) = {1, 2, 2}

<u>addN:</u>
Signature: public Bag addN(D elt, int n);
Params: D elt, int n
Returns: Bag
Description: This method returns a new bag with n more instances of elt
Example: {1, 2}.add(1, 4) = {1, 1, 1, 1, 1, 2}

<u>union:</u>
Signature: public Bag union(Bag u);
Params: Bag u
Returns: Bag
Description: This method returns a new bag that unions the (Bag) instance it was called on and the input Bag u.
Example: {1, 2}.union({2, 3}) = {1, 2, 2, 3}

<u>inter:</u>
Signature: public Bag inter(Bag u);
Params: Bag u

Returns: Bag
Description: This method returns a new bag that takes the intersection of the (Bag) instance it was called on and the input Bag u.
Example: {1, 2}.inter({2, 3}) = {2}

diff:

Signature: public Bag diff(Bag u);
Params: Bag u
Returns: Bag
Description: This method takes the difference of this (Bag) object and the input Bag u
Example: {1, 2}.diff({2, 3}) = {2, 3} - {1, 2} = {1, 3}

equal:

Signature: public boolean equal (Bag u);
Params: Bag u
Returns: Boolean
Description: This method returns true if ‘this’ bag equals in the input Bag u; otherwise, it returns false
Example: {1, 2, 2}.equal({1, 2}) = false; {1, 2, 2}.equal({1, 2, 2}) = true;

subset:

Signature: public boolean subset (Bag u);
Params: Bag u
Returns: Boolean
Description: This returns true if this object is a subset of u; otherwise, it returns false;
Example: {1, 2}.subset({1, 2, 2}) = true; {1, 2, 2}. subset({1, 2}) = false

seq

Signature: public Sequence<D> seq();
Params: None
Returns: Sequence<D>
Description: This method takes ‘this’ Bag and turns it into a sequence, which it returns
Example: {1, 2, 3}.seq() = [1, 2, 3]

sumIt:

Signature: public int sumIt ();
Params: None
Returns: int
Description: This method returns the number of times it iterates through ‘this’ Bag
Example: {1, 2, 4}.sumIt() = 3

sumItS

Signature: public int sumItS(Sequence<D> as);
Params: Sequence<D> as
Returns: int
Description: This method returns the number of times it iterates through the input sequence. It is a helper function for sumIt
Example: {1, 2, 4}.sumItS([1,2,4]) = 3

stringIt

Signature: public String stringIt();
Params: None
Returns: String
Description: This method returns a string with all the elements of 'this' Bag
Example: {1, 2, 4}.stringIt() = "1 2 4"

stringItS

Signature: public String stringItS(Sequence<D> as);
Params: Sequence<D> as
Returns: String
Description: This method returns a string with all the elements of the sequence. It is a helper function for stringIt.
Example: {1, 2, 4}.stringItS([1,2,4]) = "1 2 4"

addInner

Signature: Bag<D> addInner(D elt, int n);
Params: D elt, int n
Returns: Bag<D>
Description: This method serves as a helper function for add to help balance the tree
Example: {1, 2, 4}.addInner(5, 2) = {1, 2, 4, 5, 5)

isRedHuh

Signature: public boolean isRedHuh();
Params: None
Returns: boolean
Description: This method returns true if the root is red, false if it is black
Example: {1}.isRedHuh() = false; (trees always start black)

Properties Between Functions:

The following properties hold true for these functions and multi-bags. In each case **t** is a *Bag*, **r** is a *Bag*, **x** is of type *D*, **y** is an *D*, **c** is an *int*, and **nT** is a new *Bag* that I'm creating:

A. (isEmptyHuh **t**) = true <-> **t** = empty()
B. (cardinality **t**) = 0 <-> (isEmptyHuh **t**) = true
C. (sumIt **t**) == (cardinality **t**)

D. [**c** = cardinality (remove **t x**)] <->
{ [**c** = (cardinality **t**) - 1 && (getCount **t x**) >= 1]
∨ [**c** = (cardinality **t**) && (getCount **t x**) = 0]
}
E. (member **t x**) = false && **nT** = remove (add **t x**) x <-> (equals **t nT**)
nT = (add **t x**) <-> (getCount **nT x**) - 1 == (getCount **t x**)
nT = (remove **nT x**) <-> (getCount **nT x**) == (getCount **t x**)
F. member (add **t x**) **y** = true <-> **x** = **y** ∨ (member **t y**) = true
G. member (union **t r**) **x** = true <-> (member **t x**) = true ∨ (member **r x**) = true
(member (union **t r**) **x**) && (member **t x**) <-> (getCount **t x**) > 0
(member (union **t r**) **x**) && (member **r x**) <-> (getCount **r x**) > 0
!(member (union **t r**) **x**) || !(member **t x**) || !(member **r x**) <->
(getCount **t x**) == 0 &&
(getCount (union **t r**) **x**) == 0 &&
(get Count **r x**) == 0

H. (getCount (union **t r**) **x**) == (getCount **t x**) +(getCount **r x**)
I. (equal (diff t r) t) && (equal (diff r t) r) <->
[!(isEmptyHuh r) && !(isEmptyHuh t) —>
subset r t) == (subset t r) == false]

J. (equal (inter **t r**) empty()) <-> (equal (diff **r t**) **t**)
K. (equal t r) <-> (cardinality (union t r) == (cardinality (inter t r)) * 2
L. **nT** = (inter **t** empty()) <-> (equal empty() **nT**)

Performance Characteristics:
The bag is represented as a balanced tree so any “look up” function runs log(N) time. The other functions run linear time.

Interface Generator<D extends java.lang.Comparable>

All Superinterfaces: None

All Known Implementing Classes: BooleanGenerator, IntGen, StringGen

public interface Generator<D extends java.lang.Comparable>

Description: A generator is a generator of random comparable data.

Method Summary: returnValue methodName(dataType Param1, dataType Param2)

D nextThing(int min, int max);

Method Detail:

Note: All of these methods must be called on Generator instances.
Note: D is a generic-type
Note: In the examples, curly braces {} represent Bag data structures and brackets [ ] represent sequence data structure.

nextThing:
Signature: public D nextThing(int min, int max);
Params: int min, int max
Returns: D
Description: This method returns a new random object with size between the min and max
Example: intGen.nextThing(0, 3) = 2

Properties Between Functions:
The output of the nextThing function will always be of the same data as the generator. This helps generate new random data for a function, test, etc.

Performance Characteristics:
The nextThing function is extremely fast because it runs on a consistent time, where the input does not impact the efficiency performance of the function.

————————————————————————————————————

Interface Sequence<D extends java.lang.Comparable>

All Superinterfaces: None

All Known Implementing Classes: Sequence_Cat, Sequence_Empty, Sequence_NonEmpty

public interface Sequence<D extends java.lang.Comparable>

Description: A sequence is an iterable data structure comprised of comparable

data

Method Summary: returnValue methodName(dataType Param1, dataType Param2)

D here();
boolean hasNext();
Sequence<D> next();
String toStringSequence();

Method Detail:

Note: All of these methods must be called on Sequence instances.

Note: D is a generic-type
Note: In the examples, curly braces {} represent Bag data structures and brackets [ ] represent sequence data structure.

here:

Signature: public D here();
Params: none
Returns: D
Description: This method returns the object that the sequence is located at.
Example: [1, 2, 3], where the here field = 3. [1, 2, 3].here() = 3.

hasNext:

Signature: public boolean hasNext();
Params: none
Returns: Boolean
Description: This method returns true if the sequence is not at the end (i.e., there is another element in the sequence); otherwise it returns false
Example: emptySequence.hasNext() = false

next:

Signature: public Sequence<D> next();
Params: none
Returns: Sequence<D>
Description: This method returns the sequence following where the here (where sequence is located)
Example: [1, 2, 3], where the here field = 2. [1, 2, 3].next() = [3]

toStringSequence:

Signature: public String toStringSequence();
Params: none
Returns: String
Description: This method returns a string containing all the elements in the sequence
Example: [1, 2, 3].toStringSequence() = “1 2 3”

Properties Between Functions & Tips:
If the sequence is empty, the toStringSequence should print out nothing.
Using here, hasNext, and next, a client can iterate through the multi-set and apply a given function to each element of the set.

Performance Characteristics:
Since the iteration of a sequence through each item individually, a sequence runs on linear time.

———————————————————————————————————

Interface Sequenced<D extends java.lang.Comparable>

All Known Subinterfaces: Bag<D>

All Known Implementing Classes: Sequence_NonEmpty, Sequence_Cat,
Sequence_Empty, SetBag_Empty, SetBag_NonEmpty

public interface Sequenced<D extends java.lang.Comparable>

Method Summary: returnValue methodName(dataType Param1, dataType Param2)

public Sequence<D> seq();

Description: A Sequenced object has a method that can turn the object into a

Sequence (see Sequence above)

Method Detail:

Note: All of these methods must be called on Sequenced instances.
Note: D is a generic-type

seq:
Signature: public Sequence<D> seq();
Params: none
Returns: Sequence<D>
Description: This method takes 'this' object and turns it into a sequence, which it returns
Example: {1, 2, 3}.seq() = [1, 2, 3]

Properties Between Functions & Tips:
Any 'Sequenced' object can have its contents organized into a sequence where the client can apply any appropriate function to each element in the sequence (i.e., in the 'Sequenced' object)

Performance Characteristics:
The seq() function runs linear time because each branch & node must become a part of the sequence.