**SETS AND THE “Set” INTERFACE**

Diagram

Description automatically generated

The Set Abstraction

A set is a collection that contains no duplicate elements and at most one null element

Operations on sets include :

* testing for membership
* adding elements
* removing elements
* union A B
* intersection A B
* difference A - B
* subset A B

The Set Interface and Methods

Required methods:

* testing set membership
* testing for an empty set
* determining set size
* creating an iterator over the set

Optional methods:

* adding an element
* removing an element

Constructors to enforce the “no duplicate members” criterion

* The add method does not allow duplicate items to be inserted

Required method:

* containsAll tests the subset relationship
  + Larger set contains all the elements of the smaller set (subset method)

Optional methods:

* addAll (union)
* retainAll (intersection)
* removeAll (difference)

Table

Description automatically generated

If coll is also a set, it is like union.

Chart, bubble chart

Description automatically generatedThis will change original setA

Chart, bubble chart

Description automatically generatedWe made a copy so original setA will not change.

Chart, bubble chart

Description automatically generatedAgain, this will not change the original setA.

Text, letter

Description automatically generated

Comparison of Lists and Sets

Collections implementing the Set interface may contain only unique elements

Unlike the List.add method, the Set.add method returns false if you attempt to insert a duplicate item

Unlike a List, a Set does not have a get method -- elements cannot be accessed by index

You can iterate through all elements in a Set using an Iterator object, but the elements will be accessed in arbitrary order:

for (String nextItem : setA){

*//Do something with nextItem*

*. . .*

}

Order is unknown. Depending on the implementation, order will be different. You can only be sure that all the elements are iterated.

If you want to implement Set interface, how can you do it?

You can use simple arrays and don’t use indexes.

Efficiency:

* add(E obj) will take linear time ((n)) because we should traverse and be sure element is not in the array. If the element doesn’t exist, we add to the end at (amortized) constant time.
* In addAll(Collection<E> coll), we have to iterate through all the elements in the coll by using iterator or using a for loop. Then add every element by using add(E obj) method. If there are n elements in the coll and m elements in our set before performing addAll, and if none of the elements in coll are don’t exist in our set (worst case), insertion of the elements respectively will be:
  + m + (m+1) + (m+2) + … + (m+n)
    - m: first insertion, (m+1) will be second insertion …
* contains(Object obj) is linear time at the worst case, constant time at the best case.
* isEmpty() is constant time.
* remove(Object obj) takes linear time at the worst case bc we have to make shifting.

You can use ArrayList or LinkedList. Performance will be different than simple array.

You can use binary search tree. All methods depend on the height of the tree. Average case running time is better. It is logarithmic for add, contains, and remove methods.

Efficiency:

* add(E obj) : You can delegate bst’s add method. It takes O(n) -- we didn’t say average.
* contains(Object obj) : You can delegate bst’s contains method. Name may be different.