darray

@Walheimat

December 16, 2022

Contents

1 Etymology

A **darray**, pronounced [uh-rey][1], is a dulling sequence of elements. Its continued *use*[2] dulls the sequence until no more—and no less!—than its blunt end remains.

The phonetic similarity to **array** which generally denotes often immutable fixed-size sequences was chosen for the fact that the darray can never grow[3].

2 Summary

A **darray** is both similar and dissimilar to a given language's mutable **list** (or **vector**) implementation.

It is similar in the following regards:

[1] The d is silent.
[2] Think: mutation.
[3] Not just to go against the *Principle of Least Surprise*.

- it is an iterable sequence of arbitrary length of objects or primitives or both[4]
- it is—in a unique way—mutable, ideally supporting any operation[5] the underlying mutable list implementation does

It is dissimilar in the following regards:

- items in a darray have a **direction** from sharp to dull, the lower indices being the sharpest[6]
- it has a **blunt end** that—at least after (re-)initialization—is at a max `n-1` elements if `n` is the sequence's length[7]
- following from the prior, it may also have a **sharp end** that has 1 or—after use—fewer elements
- unless the operation is the unique `sharpen` call, it **dulls** the sequence, that is, the tip of the sharp end is removed and possibly returned
- even if the operation would otherwise not mutate the equivalent list, it will still dull and thereby potentially mutate the darray
- it can be **sharpened**; when sharpened, the sequence is reset using a copy of the elements it was initialized with

3 Specification

3.1 Initialization

A darray should support initialization ergonomics similar to those the equivalent list or vector implementation provides.

For example, since Java's `ArrayList` allows double-brace initialization, the `DarrayList` should aim to behave in similar ways.

[4] If the language allows mixing types in its list implementation.

[5] The effects being quite different, as we'll see.

[6] This is because it's generally much faster to access the head of a list than its tail.

[7] The array can never be shorter than its blunt end and no member of the blunt end is "blunter" than any other member.

3.2 Mutation

A darray needs to implement the same operations an equivalent mutable list implementation does. However, any such operation that is a mutation dulls the sequence. The number of impacted elements is never greater than those remaining in the sharp end.

Dulling means that the n sharpest elements are removed and—if the operation would otherwise return elements—returned instead. Since n can be greater than the remaining elements, elements from the blunt end may be returned this way as well.

3.2.1 Examples: Preparing Implementing Java's `AbstractList` interface

In order to get a better feel of what this means, we're going to look at some mutation scenarios if we'd like to implement a `DarrayList` in Java.

1. `clear`

   > Removes all of the elements from this list.

   This would not remove all the elements from the `DarrayList`. Instead it would remove any element remaining in the sharp end. The dull end would remain untouched.

2. `remove`

   > Removes the element at the specified position in this list.

   This will always remove and return the sharpest element instead, no matter what argument is passed as `index`. If only the blunt end remains, this should throw and `IndexOutOfBoundsException` *even if* the `index` would otherwise denote an element in the blunt end since an element of the blunt end cannot be removed.

3. `add`

   > Appends the specified element to the end of this list.

   It's not possible to *append* to a darray but just like `ArrayList` this call should return `true` if a sharp end remains and an element can be removed as it signifies the list has changed as a result.

4. `iterator`

   > Returns an iterator over the elements in this list in proper sequence.

   Remember that *any* operation potentially dulls the darray. Since we should still observe the contract of not returning outdated information, the `Iterator` returned in `DarrayList`'s case would only contain the elements remaining *after* an element was removed (if it can be).

5. `indexOf`

   > Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

   If the specified element is located in the darray, its index *after* removal should be returned. If the element happens to be the one that's broken off, we would therefore return `-1`.