Algorithms and Data structures

wk3: Double-ended singly linked list

Version: 16 September 2021.

In week 2 you have practiced the implementation of a simple double-ended singly linked list data structure with some of its methods. In this exercise you will practice a full concrete implementation of the generic data structure by extension from the AbstactList<E> class that is provided by the Java Collection Framework.

At
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/AbstractList.html
you find extensive documentation about lists in the Java Collection framework. Here we will explain the overall structure with help of a UML class diagram relating the most important classes, interfaces, and methods.

A List is a special kind of Collection that provides direct access to each of its items by means of a position index starting at zero for the first element. ArrayLists implement the List interface by storing references to the items in the consecutive memory of a regular Java array[ ]. That way the items can be addressed directly by using the position as an index in the array. LinkedLists maintain a chain of Nodes. Each Node holds a reference to the item as well as references to its successor and/or predecessor nodes. Specific items can be found by walking the chain.

The double-ended, singly linked list is a specific version of a linked list in that nodes only hold a reference to a successor node, not to the predecessor node. The list object itself has a reference to the first node in the chain (the head) and the last node in the chain (the tail).

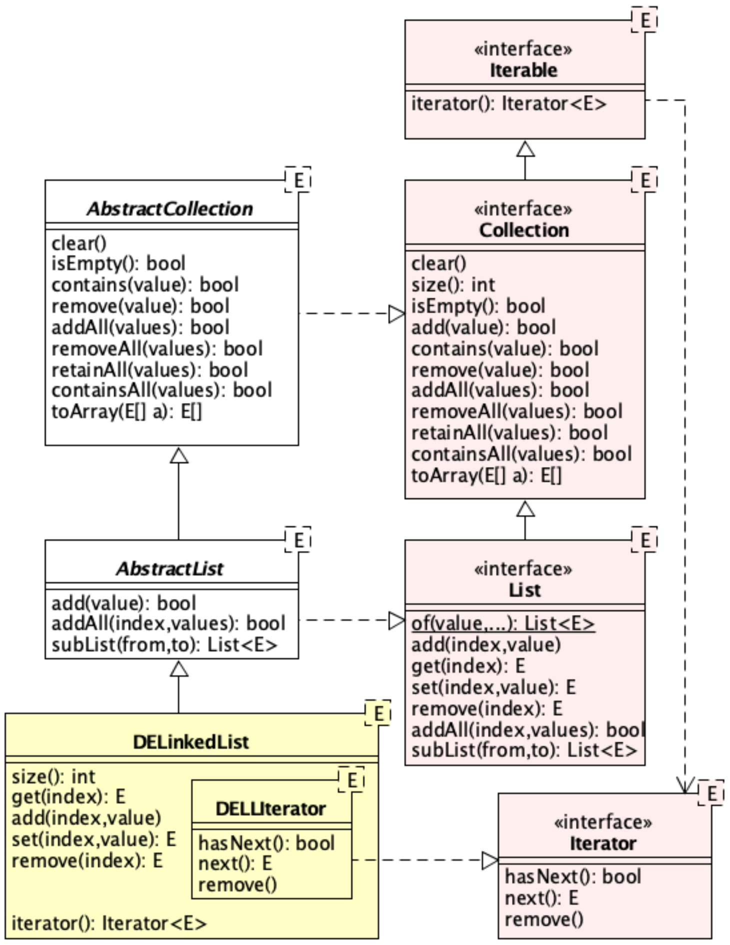
Many of the methods of the List<> and Collection<> interfaces are being implemented by the AbstractList<> and AbstractCollection<> abstract classes. You only need to implement a few basic methods yourselves and can inherit most of the methods from AbstractList<>. From the collection documentation (and as indicated in the diagram) it appears that you only need to implement:

size(),	: returns the number of items in your list.
add(index, item)	: inserts an item at given position index.
get(index)	: returns the item at given position index.
set(index, item)	: replaces the item at given position index by the given item.
remove(index)	: removes the item at given position index.
iterator<>()	: initializes and returns an iterator set-up to iterate along all items in your list.

You may want to implement some useful private helper methods in your Node class to support these implementations.
The abstract superclasses will heavily rely on your iterator to implement the other methods.

Exercise:

1. Unpack and open w3-exercise2-DELinkedList-starter in your (IntelliJ) development environment and import project configuration from the maven pom.xml file.
2. The start project provides:
   DELinkedList.java: a skeleton of the implementation of a double ended linked list, which extends AbstractList<>
   DELinkedListMain.java: a starter program running a very limited test
   test/java/ListImplementationTest.java: an extensive unit test of your implementation. With a simple change in the setup method of this test class, you can also run it on other list implementations of the framework.
   test/java/ListPerformanceTest.java: a simple performance test class which you can use to compare specific performance characteristics of list implementations.
3. Finish and test your DELinkedList implementation.
4. Document the representation invariant of your implementation.
5. Compare performances of your list implementation and the framework implementations. Below you find some expected results.



Example Performance Results:	ArrayList,	Vector,	LinkedList,	DELinkedList
✓ p01_appendPerfomanceTest()	214 ms	188 ms	168 ms	158 ms
✓ p02_prependPerfomanceTest()	1 s 198 ms	1 s 80 ms	58 ms	46 ms
✓ p03_randomAddRemovePerformanceTest()	29 ms	26 ms	735 ms	1 s 274 ms
✓ p04_randomGetSetPerformanceTest()	10 ms	14 ms	205 ms	279 ms