Structured Programming

COMP1110/6710

Australian National University

Needs ANU Account!

pollev.com/albertofmartin963
Register for Engagement

Image Courtesy NASA/JPL-Caltech.

Abstract Data Types (ADTs)

Australian National University

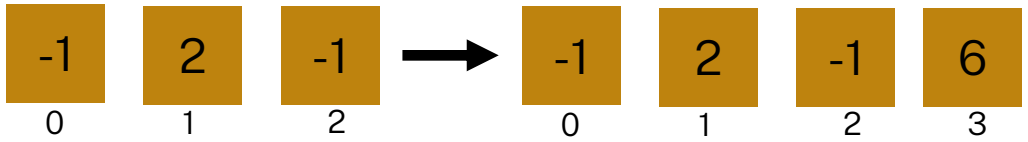
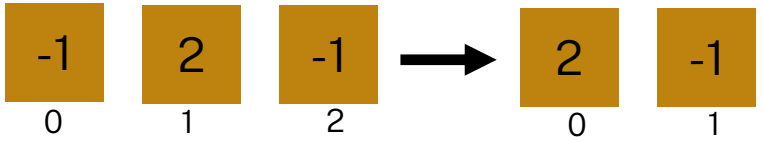
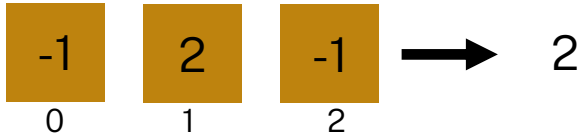
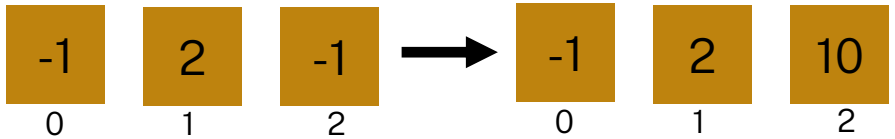
Abstract Data Types (ADTs)

- **Abstract Data Types** (ADTs) are mathematical models for data types
- They are defined in terms of **behaviour** *(semantics)* from the user's perspective, **NOT** in terms of implementation (i.e., the specific details)
- (This is why they are abstract, i.e., not concrete).
- ADTs specify (1) possible values the data type may have (but NOT how they are laid out/stored in memory), (2) a set of operations that a user may perform on the data type (but NOT how they are implemented), and (3) behaviour (semantics) of such operations
- ADTs are a **CORNERSTONE** for code reusability and modularity (as they enable code that relies on abstractions rather than concretions/specifics)
- Examples of ADTs**: List, Stack, Queue, Map, Tree, Graph, etc.**
- Typically, different programming languages offer different mechanisms to specify ADTs and their implementations

ADT example: List

- For the purpose of this workshop, we consider the list ADT to represent a finite sequence of elements of the same type (e.g., list of strings)
- A list is an example of **container** ADT, as it holds other objects (e.g., dates)
- **Fundamental properties of the List ADT:** (1) duplicate elements are allowed; (2) order of elements is preserved (thus, we can assign a unique index to each element in the list).
- Examples of list operations: (1) Create a new empty list. (2) Add a new element to the end of the list. (3) Remove first occurrence of an element from the list given the element. (4) Get the element of the list at the specified index. (5) Replace the element of the list at the specified index by the given element.

ADT example: List

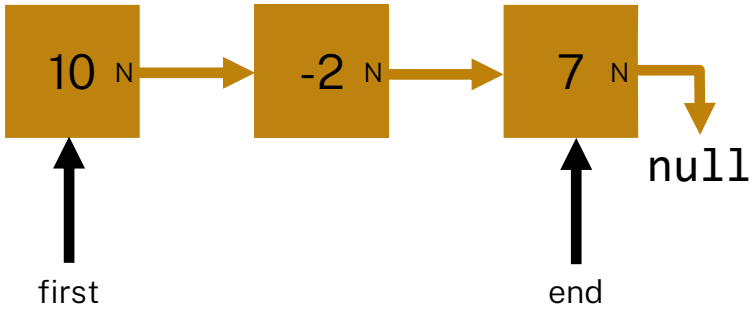
Example operation	Behaviour (semantics)
Add element 6	
Remove element -1	
Get element at index 1	
Set index 2 to 10	

ADT implementations

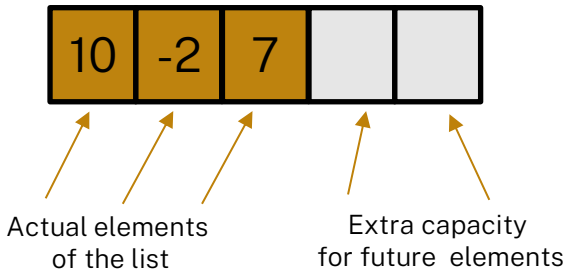
- ADTs can typically be **realized** (implemented) in different ways. A realization of an ADT is called an ADT implementation
- Any ADT implementation **MUST** adhere to the specified behaviour of the ADT and its operations (following the Liskov Substitution principle)
- However, implementations have the freedom to internally store/lay out in memory the elements differently, and thus to implement the ADT operations accordingly to the chosen internal data storage layout
- Each ADT implementation typically leads to a different trade-off/balance among memory consumption and the computational cost of its operations
- Which ADT implementation will be best for a particular algorithm will depend on the particular features of the algorithm at hand (this is why several implementations of an ADT exist)

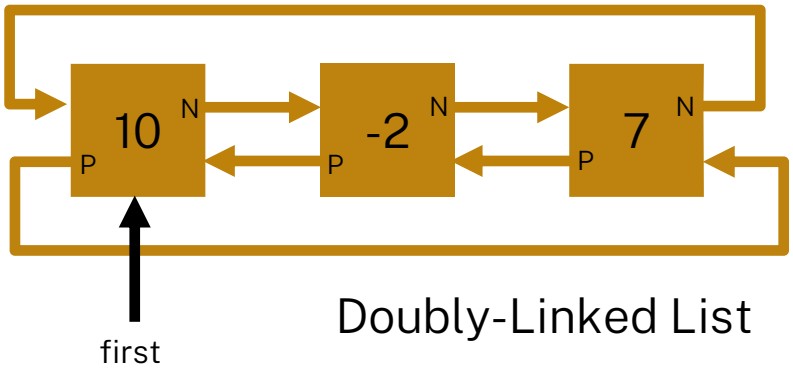
Examples of ADT implementations

Singly-Linked List



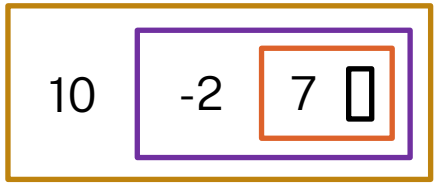
Array-based List





Doubly-Linked List

`ConsList`-based List



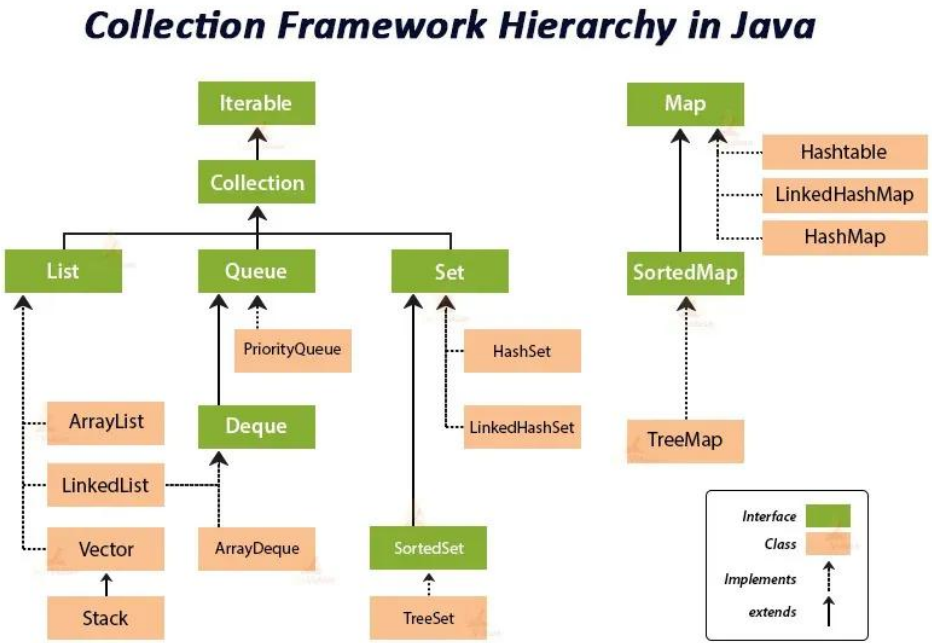
ADTs in Java

Australian
National
University

ADTs in Java

- Java offers different programming mechanisms that might be used to express ADTs and their implementations
- Among these, we will use parameterized (i.e., generic) interfaces to specify ADTs and parameterized classes implementing such interfaces for ADT implementations (as, among others, this is the approach followed by the Java standard library)
- Although we could define our own interfaces, we will leverage those in the Java standard library (so that we do not reinvent the wheel, allowing our class implementations to reuse code available in the Java standard library and beyond)
- The so-called *Java collections framework* offers, among others, a hierarchy of parameterized interfaces and class implementations for such interfaces
- **Note**: the full details regarding generics in the particular context of interfaces, classes and subtyping will be covered in next week's workshop

The Java collections framework at a glance



Source: freeCodeCamp

- Hierarchy of interfaces (green) and class implementations (light pink)
- All interfaces and class implementations are at least parameterized by the element type `E` (not shown in the picture)
- In this workshop, we will use the `List<E>` and `Collection<E>` interfaces (shown in the figure) and the `Iterator<E>` interface (not shown in the figure)
- More exploration of the framework in future workshops

The Java's `List<E>` generic interface

- Represents the List ADT (full documentation available [here])
- A simplified/partial definition of List<E> generic interface is as follows:

Interface inheritance

```
public interface List<E> extends Collection<E> {
    boolean add(E element);          // doc [here]
    boolean remove(Object o);        // doc [here]
    E get(int index);                // doc [here]
    E set(int index, E element);     // doc [here]
    int size();                      // doc [here]
    boolean isEmpty();               // doc [here]
    … // More interface method definitions
}
```

(A subtle detail here is that `List<E>` actually extends `SequencedCollection<E>`, which in turn extends `Collection<E>`)

The Java's `List<E>` generic interface

Other examples of `List<E>` methods include:

```
public interface List<E> extends Collection<E> {
    … // Methods in the previous slide
    boolean contains(Object o);                 // doc [here]
    boolean containsAll(Collection<?> c);       // doc [here]
    void clear();                               // doc [here]
    void add(int index, E element);             // doc [here]
    E remove(int index);                        // doc [here]
    Iterator<T> iterator();                     // doc [here]
    … // More interface method definitions
}
```

This is known as the unbounded wildcard in Java generics.

In a nutshell, it states that "any type" is possible as the type parameter of the Collection

More details next week

More details on the `Iterator<T>` interface on the next slide

The Java's `Iterator<E>` generic interface

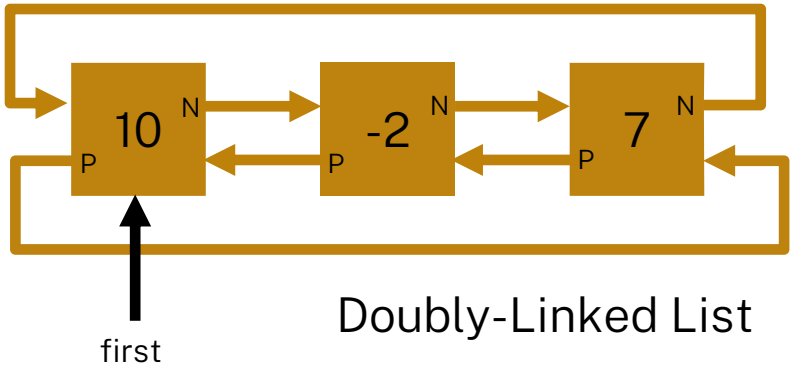
- An iterator is an object that lets one walk in sequence through the objects in a collection (without exposing the specific details of how these are stored within)
- This kind of operation comes up so frequently that the standard Java library offers a generic interface for it, namely `Iterator<E>`
- A simplified/partial definition of `Iterator<E>` generic interface is as follows:

```
public interface Iterator<E> {
    boolean hasNext();              // doc [here]
    E next();                       // doc [here]
    … // More interface method definitions
}
```

- It is typically useful to implement iterators as inner classes (although not strictly necessary; an example of this coming next) of the collection they iterate over (so that they have easy access to the collection fields)
- Iterators created out of classes implementing the `Collection<E>` interface, can be automatically used in "enhanced" for statements

DoublyLinkedList<E>

Let us write our own class implementation of the `List<E>` generic interface using the concept of a doubly-linked list of nodes



Doubly-Linked List

Practice

Design a **generic** class `DoublyLinkedList<E>`, parameterized by the type of the elements `E`, that implements a partial set of the methods in the `List<E>` interface, namely: (1) **add** a new element; (2) **remove** an element; (3) **size**; (4) **isEmpty**; (5) **get**; 6) **set**; (7) **containsAll**; (8) **iterator**. Follow the design recipe!

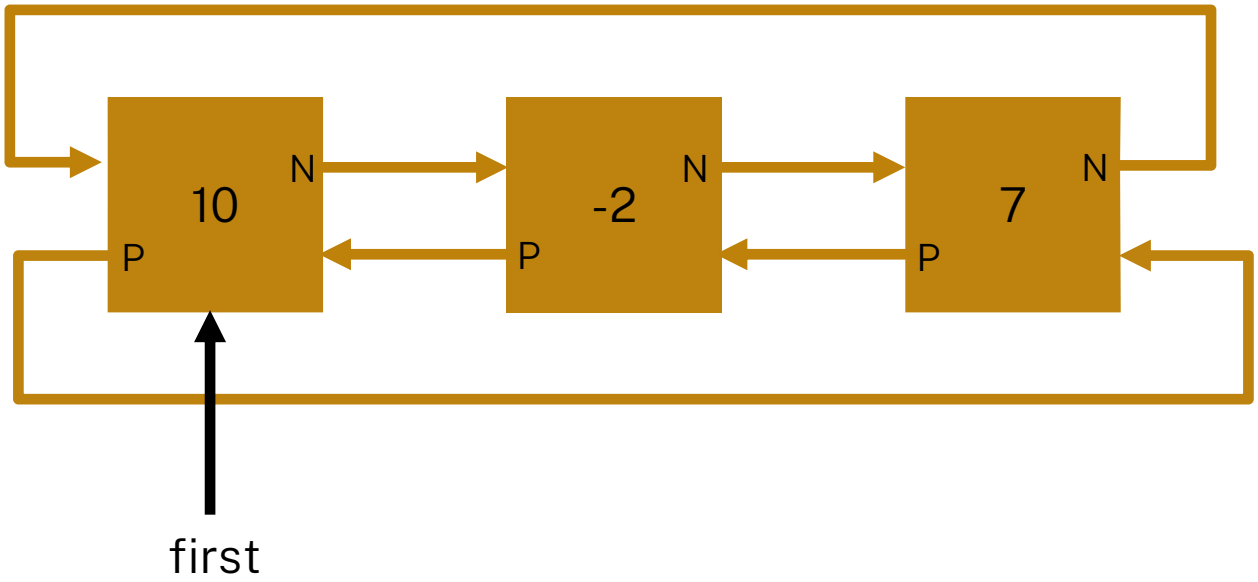
In order to get you started, we will demonstrate the class definitions and the implementation of (1) and (8). Then, you will have time to implement during the workshop the rest of operations, starting with operation (2) for which some illustrations are given below in the following slides.

DoublyLinkedList<E> **add** method illustrations

Australian
National
University

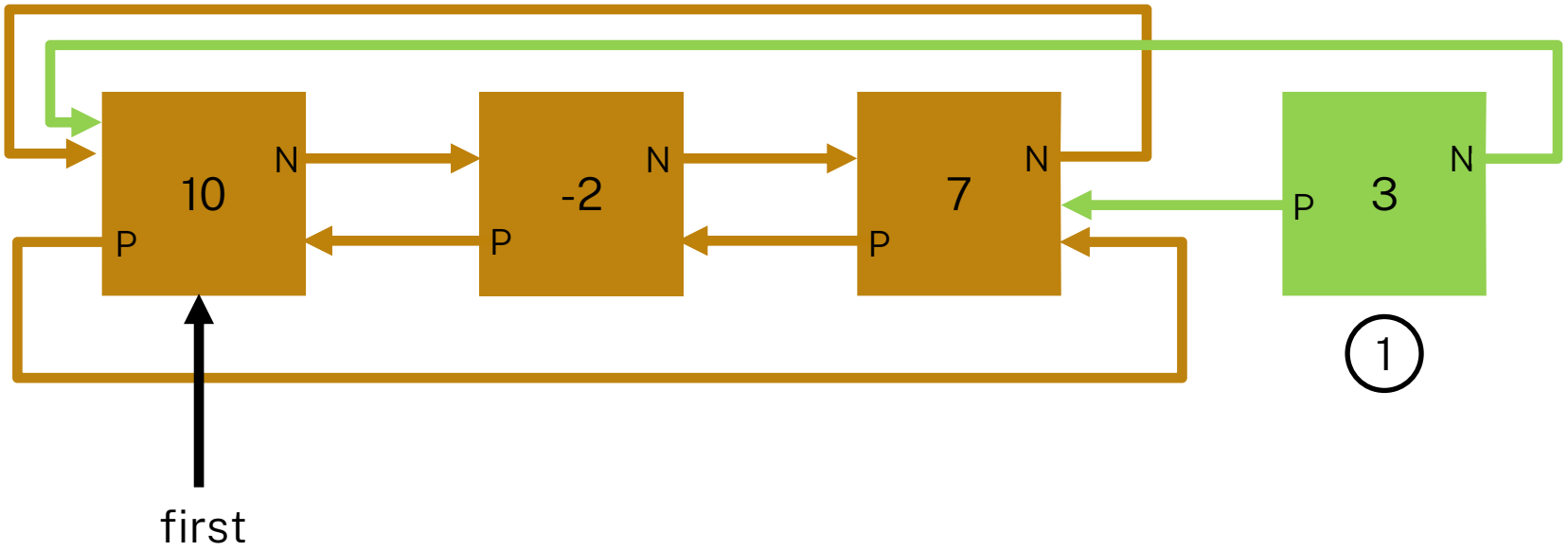
Doubly Linked List Implementation (**add** method example)

```
l.add(3)
```



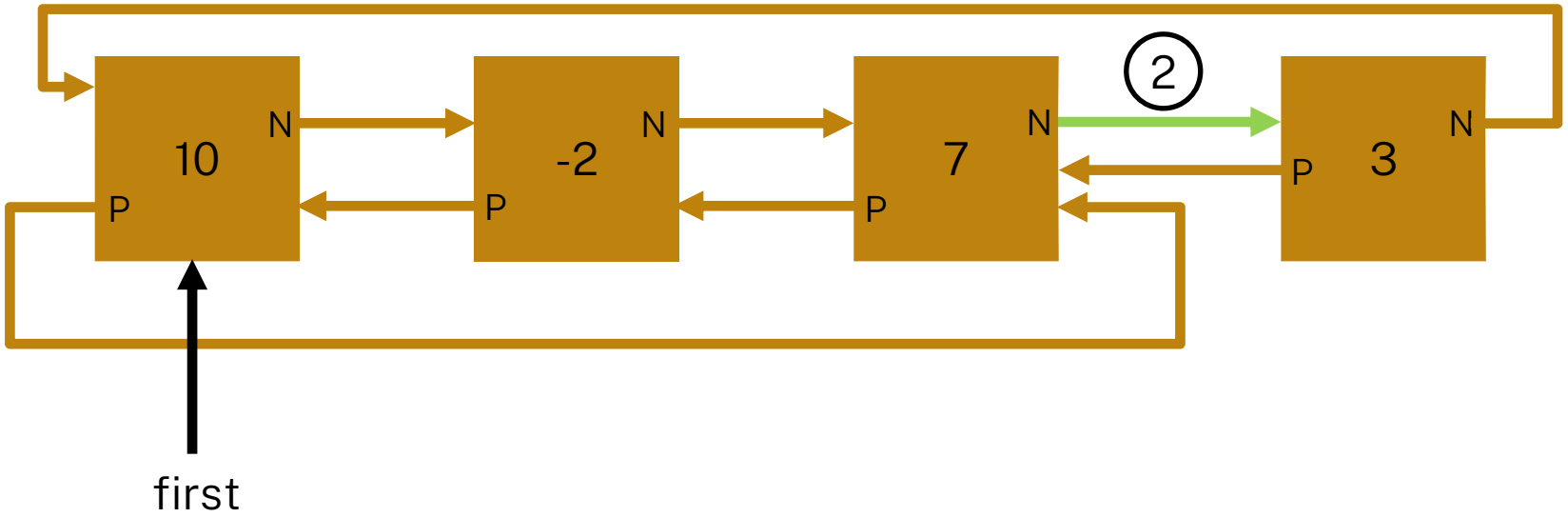
Doubly Linked List Implementation (**add** method example)

```
l.add(3)
```



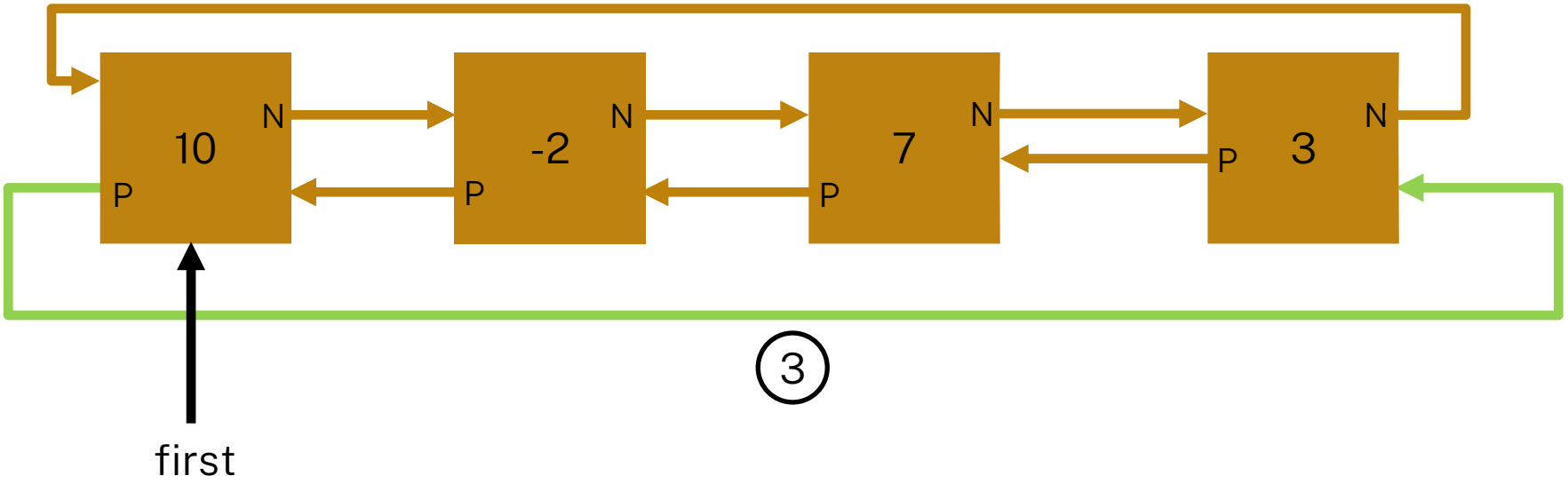
Doubly Linked List Implemention (**add** method example)

```
l.add(3)
```



Doubly Linked List Implementation (**add** method example)

```
l.add(3)
```

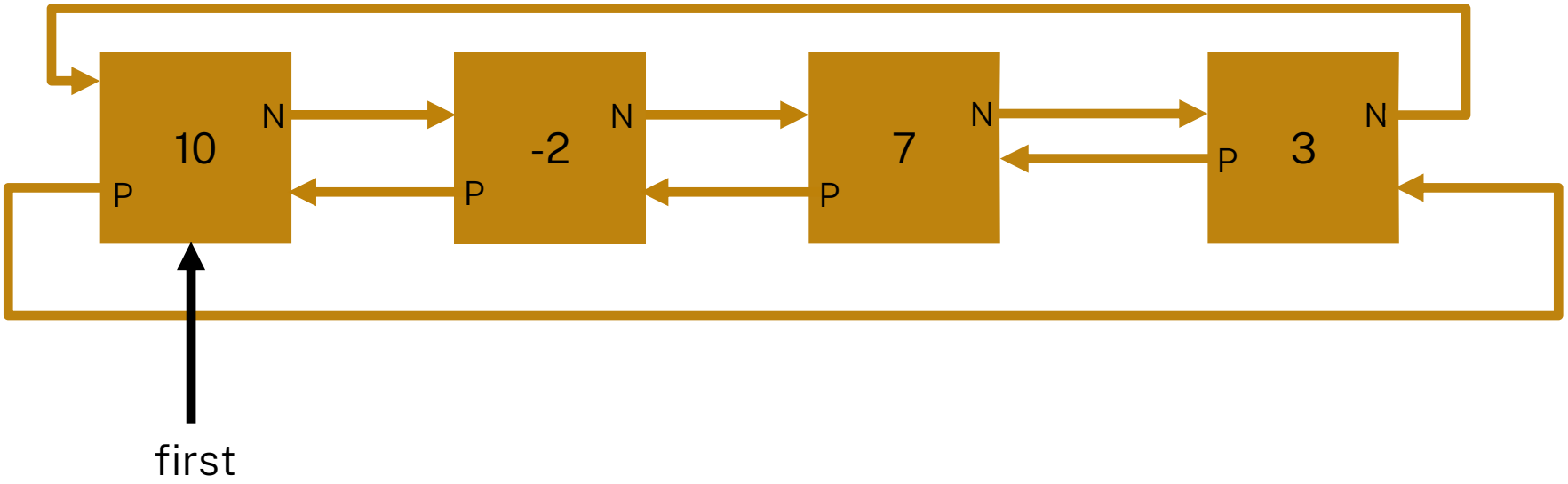


DoublyLinkedList<E> **remove** method illustrations

Australian
National
University

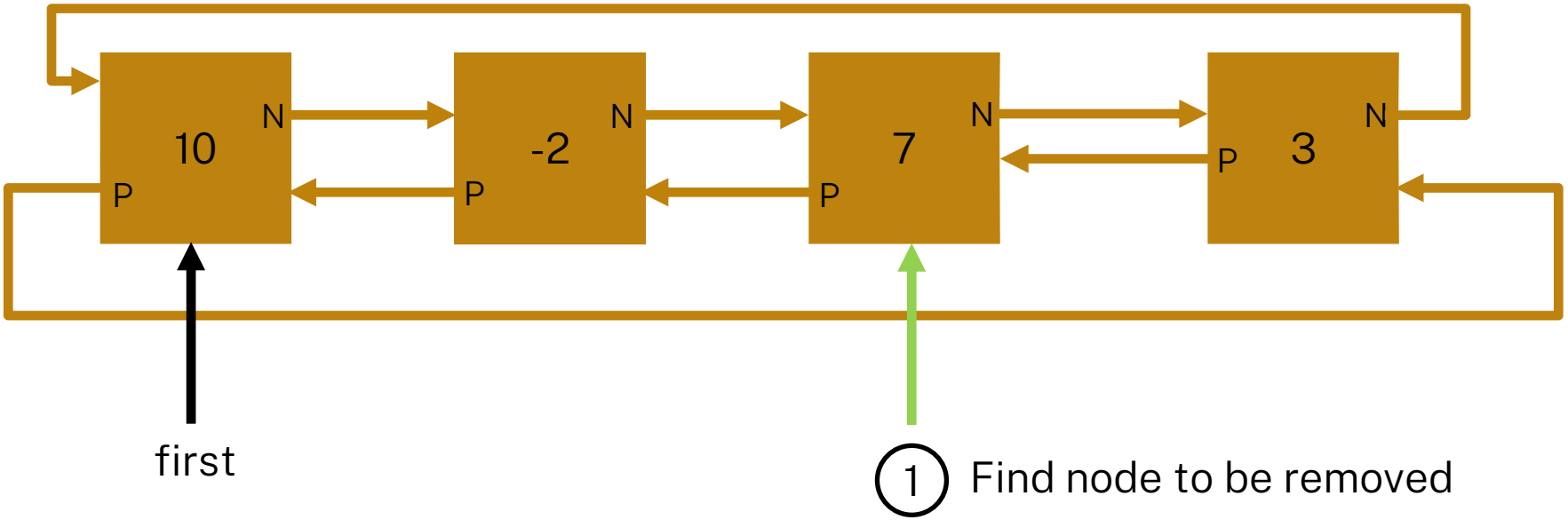
Doubly Linked List Implementation (**remove** method example)

```
l.remove(7)
```



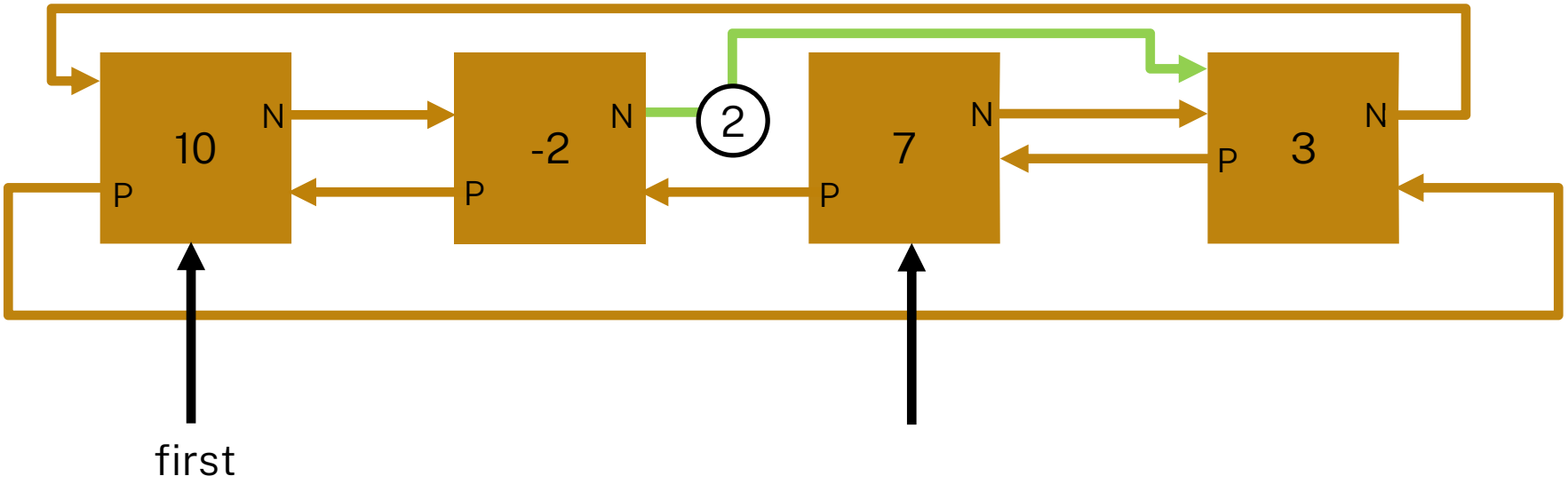
Doubly Linked List Implementation (**remove** method example)

```
l.remove(7)
```



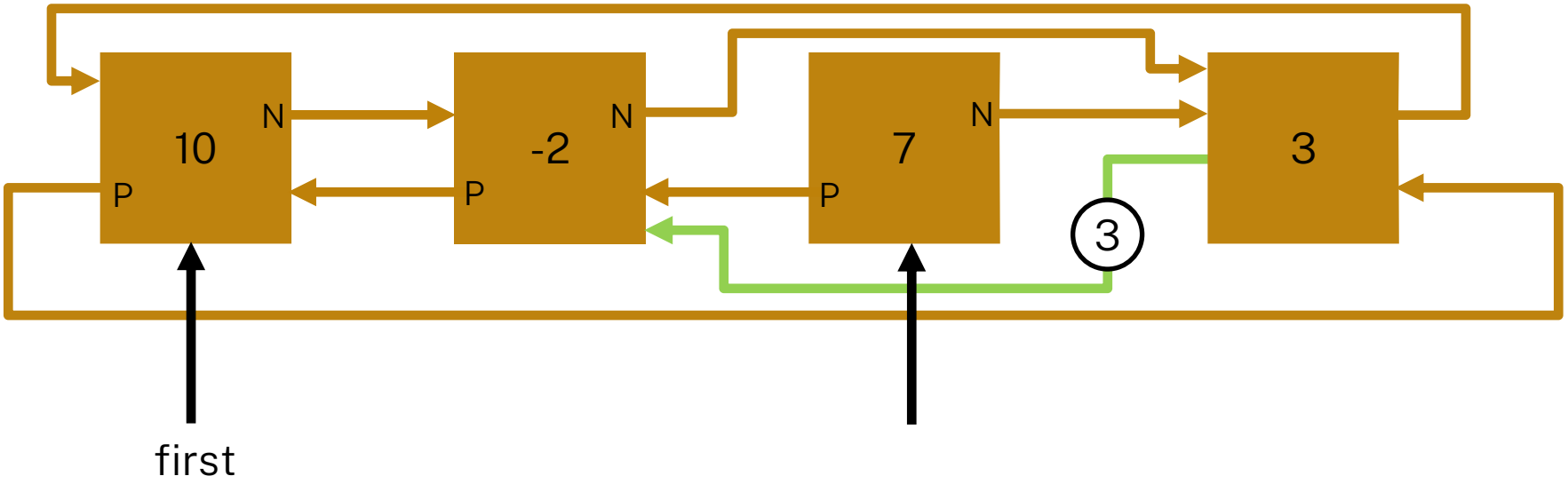
Doubly Linked List Implementation (**remove** method example)

```
l.remove(7)
```



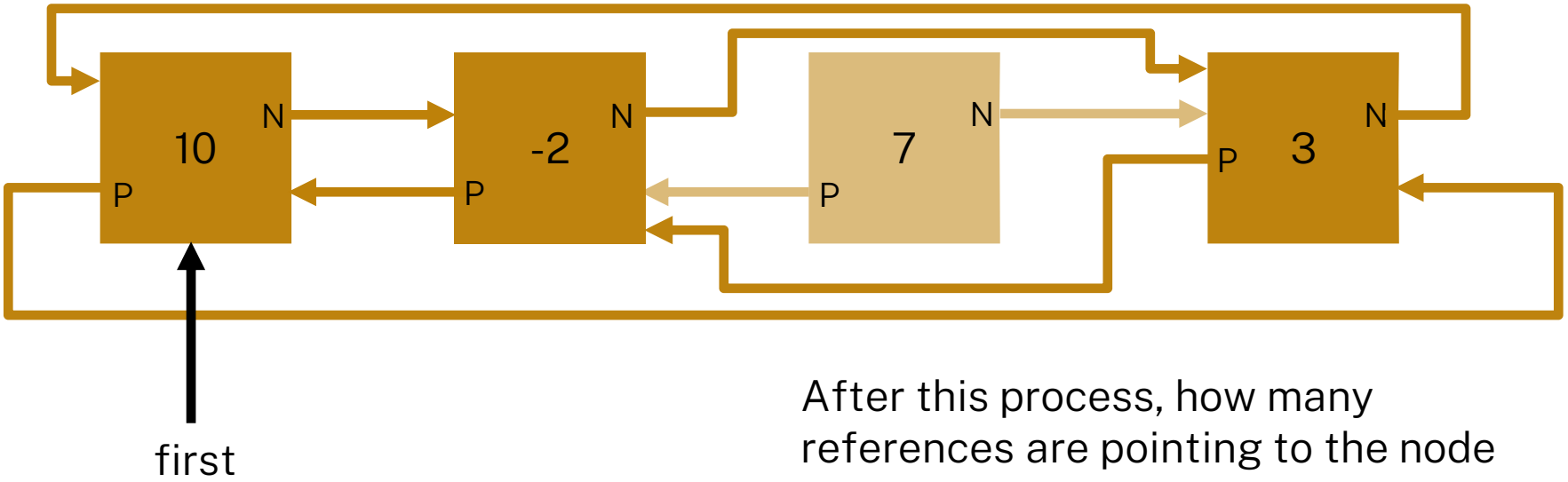
Doubly Linked List Implementation (**remove** method example)

```
l.remove(7)
```



Doubly Linked List Implementation (**remove** method example)

```
l.remove(7)
```



After this process, how many references are pointing to the node that stores the element 7?