Exercise 04: Iterators

In this exercise, we will be implementing iterators for the Stack and Set data structures from the last two exercises.

Task 1: Iterator for Set

We have a generic implementation for the set through the `SetImpl` class, following the example of the sample solution to Exercise 3.

- Implement the prescribed `iterator` method through the `Iterable` interface.
- When doing so, implement an iterator that visits all elements.
- Make sure that the `TestSet.testStringSet` test case runs correctly.

Information:

- Using an agenda is absolutely necessary, so you must somehow remember what has already been output. You can also use the stack for this purpose, for example.
- What will change in the order of visits if you use a stack for the agenda instead of a list?

Task 2: Reverse iterator for `Stack<T>`

We have a generic implementation of a stack, following the example of the sample solution to Exercise 3. A stack is a data structure whose access/removal order (pop) is reversed from the addition order (push). This means that the element added last with push is the one that is removed first with pop (LIFO: last in, first out).

- Implement (only) the prescribed `iterator` method through the `Iterable` interface. When doing so, implement a reverse iterator that visits the elements in reverse order. For a stack, this means: in the order in which the elements were added, i.e. FIFO: first in, first out.
- Make sure that the `StackTest.testStack` test case runs correctly.

An example: if the values 1, 4, 5, 2 are pushed into a Stack<Integer>, the sequence for pop is 2, 5, 4, 1. On the other hand, if we iterate over it, the order they were added should be adhered to and the values 1, 4, 5, 2 should be output again.

Information:

- The iteration of a sequential data structure can also be realised with an agenda; how does this behave over the lifetime (of the iteration)?
- Which data structure is suitable for inverting the sequence?

Task 3: Leaf iterator

Implement the `SetImpl<T>.leafIterator` method.

- Implement this iterator so that only leaf nodes are returned, i.e. elements without successors.
- Make sure that the SetTest.testLeafIterator test case runs correctly.

Information:

This task is extremely tricky! The main difficulty is ensuring that the `hasNext` method also works reliably. One approach is to also maintain a reference to the next leaf, in addition to the agenda, for traversing the tree structure. If the next element is requested, we can return it quickly, but we must thereafter try to find the next element immediately.