POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Uwaga: To jest zadanie-killer. Jak ktoś je dobrze (i samodzielnie) ogarnie, to może spać spokojnie mając świadomość, że na pewno dobrze napisze kolokwium i egzamin z PPJ. Czas przewidziany na realizację tego zadania: 15 godzin.

Zadanie

Zadanie polega na stworzenia samosortującej się, odwracalnej listy podwójnie wiązanej (ang. *doubly-linked list*), która będzie przechowywała studentów posortowanych po ich nazwiskach, imionach i eSkach.

W ramach tego zadania wpierw przygotuj klasy:

- `Student`, złożoną z pól `String name`, `String surname` oraz `int s` (może być `record`).
- `Node`, złożoną z pól `Student student`, `Node next` oraz `Node previous` (**nie** `record`).
- `SortedDoublyLinkedList`, złożoną z pól `Node head` i `Node tail` (`head` reprezentuje pierwsze ogniwo (ang. *node*), a `tail` ostatnie) (**nie** `record`).

Obiekty typu `SortedDoublyLinkedList` powinny wspierać (pośrednie) przechowywanie obiektów typu `Student` (w końcu składają się z powiązanych ogniw, które przechowują obiekty typu `Student`). Gdy dodawany jest nowy student do listy, powinien on zostać umieszczony w odpowiednim miejscu (niekoniecznie na końcu). Odpowiednie miejsce to takie, gdzie przechodząc po wszystkich ogniwach od `head` do `tail`, natrafimy na studentów uporządkowanych alfabetycznie względem nazwiska.

Każdy `Node` wie o elemencie go poprzedzającym (dzięki `.previous`) oraz elemencie po nim następującym (dzięki `.next`).

Przykładowo, następujący kod:

```
1  SortedDoublyLinkedList l = new SortedDoublyLinkedList();
2
3  l.add(new Student("Michał", "Tomaszewski", 534));
4  l.add(new Student("Sławomir", "Dańczak", 12764));
5  l.add(new Student("Filip", "Kwiatkowski", 17137));
6
7  System.out.println(l);
```

Powinien wyświetlić:

```
1  (Sławomir Dańczak 12764)->(Filip Kwiatkowski 17137)->(Michał Tomaszewski
   534)
```

To dlatego, że ustawiając tych studentów alfabetycznie po nazwiskach, otrzymamy kolejność `Dańczak` < `Kwiatkowski` < `Tomaszewski`.

Gdyby jednak utworzyć dwóch studentów o identycznych nazwiskach, ale innych imionach, ich relatywna kolejność powinna być zależna od kolejności alfabetycznej imion:

```
SortedDoublyLinkedList l = new SortedDoublyLinkedList();

l.add(new Student("Michał", "Tomaszewski", 534));
l.add(new Student("Sławomir", "Dańczak", 12764));
l.add(new Student("Filip", "Kwiatkowski", 17137));
l.add(new Student("Adam", "Dańczak", 1337)); // nowy element

System.out.println(l);
```

```
(Adam Dańczak 1337)->(Sławomir Dańczak 12764)->(Filip Kwiatkowski 17137)->(Michał Tomaszewski 534)
```

Oczywiście w sytuacji, gdy zarówno imię jak i nazwisko są identyczne, liczy się numer `s`. Gdy wszystkie pola są identyczne, to przecież i tak nie rozróżnimy tych obiektów, a zatem można je dodać gdziekolwiek obok siebie.

Dodatkowo, jak widać, klasa `SortedDoublyLinkedList` odpowiednio nadpisuje metodę `.toString()`, dzięki czemu taka lista jest wyświetlana w przystępny sposób. Warto zwrócić uwagę, że w takim przypadku wypadałoby również nadpisać `.toString()` dla `Node` i `Student`, aby `.toString()` dla naszej listy było łatwiejsze do zaimplementowania.

Nasze listy powinno się dać odwracać:

```
SortedDoublyLinkedList l = new SortedDoublyLinkedList();

l.add(new Student("Michał", "Tomaszewski", 534));
l.add(new Student("Sławomir", "Dańczak", 12764));
l.add(new Student("Filip", "Kwiatkowski", 17137));
l.add(new Student("Adam", "Dańczak", 1337));

System.out.println(l);
l.reverse();
System.out.println(l);
```

```
(Adam Dańczak 1337)->(Sławomir Dańczak 12764)->(Filip Kwiatkowski 17137)->(Michał Tomaszewski 534)
(Michał Tomaszewski 534)->(Filip Kwiatkowski 17137)->(Sławomir Dańczak 12764)->(Adam Dańczak 1337)
```

Należy jednak zwrócić uwagę, że logika dodania studentów opiera się o wstawienie ich w odpowiednie miejsce (wynikające z ustalonej kolejności pól). Po odwróceniu listy, dodanie nowego studenta powinno „wiedzieć", że lista jest w „stanie odwrócenia". To znaczy, że gdyby teraz dodać nowego studenta:

```
l.add(new Student("Jan", "Zamoyski", 420));
```

To powinien on być dodany **na początku**. To dlatego, że mimo iż jego nazwisko jest alfabetycznie „ostatnie”, to lista została odwrócona, więc teraz reprezentuje elementy „od końca do początku”.

Jeżeli lista zostanie odwrócona ponownie, to jej „stan odwrócenia” wraca do pierwotnego, czyli znowu elementy (dotychczas przechowywane i później dodawane) będą ustawiane „rosnąco”.

Obiekty typu `SortedDoublyLinkedList` powinny też wspierać informowanie o tym, ile przechowują w środku elementów przy pomocy metody `.size()`.

Obiekty te powinny również wspierać usuwanie pierwszego elementu, ostatniego elementu i elementu o dowolnym indeksie:

```
1  SortedDoublyLinkedList l = new SortedDoublyLinkedList();
2
3  l.add(new Student("Adam", "Dańczak", 1337));
4  l.add(new Student("Sławomir", "Dańczak", 12764));
5  l.add(new Student("Filip", "Kwiatkowski", 17137));
6  l.add(new Student("Michał", "Tomaszewski", 534));
7
8  l.removeAt(2);
9  l.removeFirst();
10 l.removeLast();
11
12 System.out.println(l);
```

```
1  (Sławomir Dańczak 12764)
```

Oczywiście należy pamiętać, że niezależnie od kolejności wykonania metod `.add()` między linijkami `3-6`, wynik powinien być taki sam (bo lista sama się sortuje).

Ostatnim elementem zadania jest stworzenie polimorficznej hierarchii obiektów konsumujących elementy z naszej listy. Klasą bazową ma być `Consumer` z metodą `.accept()`, która przez argument przyjmie obiekt typu `Student`.

Pierwszą z klas dziedziczących po `Consumer` ma być `ArrayBuilder`, który stworzy tablicę na podstawie elementów przekazanych do niego przez obiekt typu `SortedDoublyLinkedList`. Lista ma dostarczyć elementy do konsumerów poprzez metodę `.supplyTo()`:

```
1  SortedDoublyLinkedList l = new SortedDoublyLinkedList();
2
3  l.add(new Student("Filip", "Kwiatkowski", 17137));
4  l.add(new Student("Michał", "Tomaszewski", 534));
5  l.add(new Student("Sławomir", "Dańczak", 12764));
6
7  ArrayBuilder builder = new ArrayBuilder();
8  l.supplyTo(builder);
9  System.out.println(Arrays.toString(builder.getStudents()));
```

```
1  [Sławomir Dańczak 12764, Filip Kwiatkowski 17137, Michał Tomaszewski 534]
```

Drugą z klas dziedziczących po `Consumer` ma być `IndexFinder`, który pamięta **ostatniego** studenta z listy, który ma równy numer indeksu do tego, którego sprecyzujemy obiektowi w konstruktorze:

```
SortedDoublyLinkedList l = new SortedDoublyLinkedList();

l.add(new Student("Filip", "Kwiatkowski", 11));
l.add(new Student("Michał", "Tomaszewski", 11));
l.add(new Student("Sławomir", "Dańczak", 999));

IndexFinder finding11 = new IndexFinder(11);
l.supplyTo(finding11);
System.out.println(finding11.getTrackedStudent());
```

```
Michał Tomaszewski 11
```

Trzecią i czwartą klasą w tej hierarchii mają być `MinFinder` oraz `MaxFinder`, które będą zapamiętywały kolejno najmniejszego i największego studenta w ramach numeru indeksu:

```
SortedDoublyLinkedList l = new SortedDoublyLinkedList();

l.add(new Student("Filip", "Kwiatkowski", 12));
l.add(new Student("Michał", "Tomaszewski", 11));
l.add(new Student("Sławomir", "Dańczak", 10));

MinFinder minFinder = new MinFinder();
MaxFinder maxFinder = new MaxFinder();

l.supplyTo(minFinder);
l.supplyTo(maxFinder);
System.out.println("Min index: " + minFinder.getTrackedStudent());
System.out.println("Max index: " + maxFinder.getTrackedStudent());
```

```
Min index: Sławomir Dańczak 10
Max index: Filip Kwiatkowski 12
```

Jak widać, elementy z hierarchii klas `Consumer` posiadają też dodatkowe metody i / lub pola (np. `.getTrackedStudent()` czy `.getStudents()`; należy jednak pamiętać, że nie wszystkie klasy mają takie same metody). Należy je dostarczyć tak, aby powyższe przykłady kodu się kompilowały i zwracały wynik zgodny z zaprezentowanym. Oczywiście biorąc pod uwagę fakt, że zakładamy, że hierarchia `Consumer` jest polimorficzna, to klasa `SortedDoublyLinkedList` nie powinna oferować żadnych przeciążeń metody `.supplyTo()` rozróżniających, jaki `Consumer` jest przekazywany. Wystarczy pojedyncza implementacja przyjmująca jako argument `Consumer`.

Dodatkowe uwagi:

- Metoda `.reverse()` z klasy `SortedDoublyLinkedList` powinna być zrealizowana **rekurencyjnie**.
- Gdy próbujemy usunąć element przy pustej liście, powinien zostać podniesiony wyjątek `RemovalFromEmptyListException`.

- Gdy próbujemy usunąć element o nieistniejącym indeksie, powinien zostać podniesiony wyjątek `IndexOutOfBoundsException` z wiadomością `"Index # is out of bounds for list of size #"`, gdzie pierwszy `#` to przekazany indeks, a drugi `#` to liczba elementów w naszej liście.

🔥 **GLHF** 🔥