Goal

Implement merge sort in place of generic linked lists, and apply it to calculating the word distribution of a text (function that returns the number of occurrences of a word in a text), and to calculating a median value of a set of numeric values.

Source recovery

Extract the contents of the `src.zip` file to the project's `src` directory. Extract the contents of the `text.zip` file to the project's root directory.

Enable assert in your Java virtual machine

In *Visual Studio Code*, type the key combination `[Ctrl]+[,]` (the second key is 'comma'), enter *vm args* in the search bar. The item `Java>Debug>settings: Vm Args` appears with a text bar in which you must write `-ea` (for enable asserts).

As before. Submit the `HW6.java`

1 Linked lists

The class `Singly<E>`, written in the file `HW6.java`, implements linked lists of objects of class `E` . We choose to represent the empty list by the value `null`. Each object has two public fields:

- `E element`, the content of a cell in the list;
- `Singly<E> next`, the rest of the list.

The class `Singly<E>` offers a constructor

- `Singly(E element, Singly<E> next)`

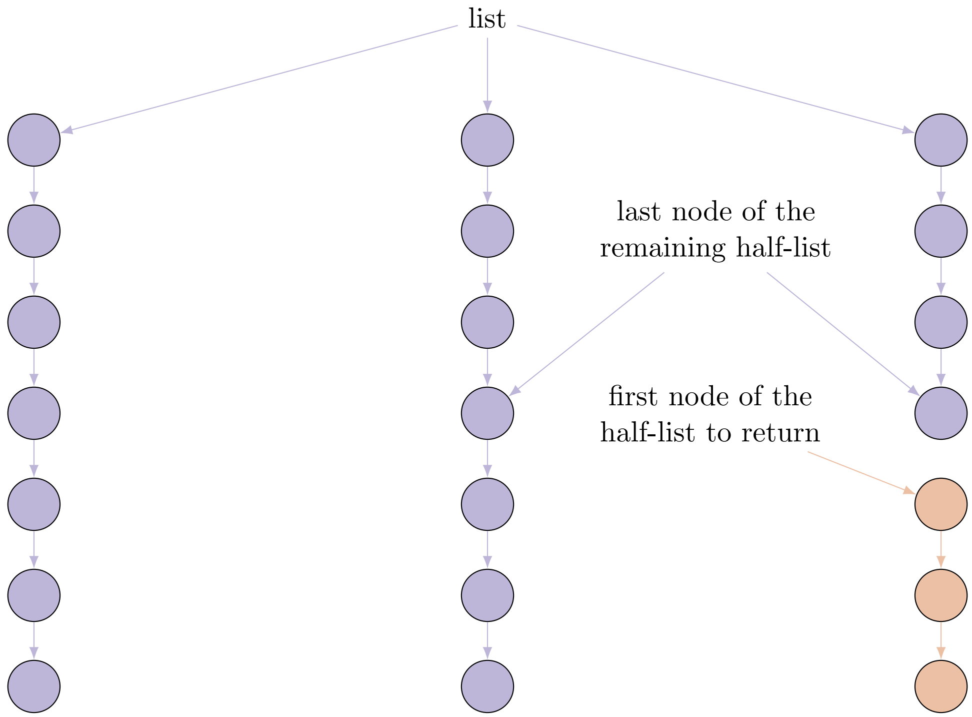
as well as other methods used in the tests, which you can ignore.

Question 1: Complete the methods

- `static<E> int length(Singly<E> l)` which returns the length of its argument.
  Note: to avoid a stack overflow (i.e. `StackOverflowError`), we will write a loop rather than a recursive method.
- `static<E> Singly<E> split(Singly<E> l)` which cuts the *non-empty* list `l` into two equal halves and returns the second half. In particular, the size of the list `l` passed as argument is halved. By convention, if the length of the string is odd, the first half contains one more element than the second.

The `split` method works in place: we go through half of the list and return the contents of the `next` field of the element we stopped on, while setting the value of this same field to `null` , which has the effect of "cutting" the list.

Figure 1


list
last node of the remaining half-list
first node of the half-list to return

Here again, we will write a loop rather than a recursive method.

Test your code by running the `Test1.java`.

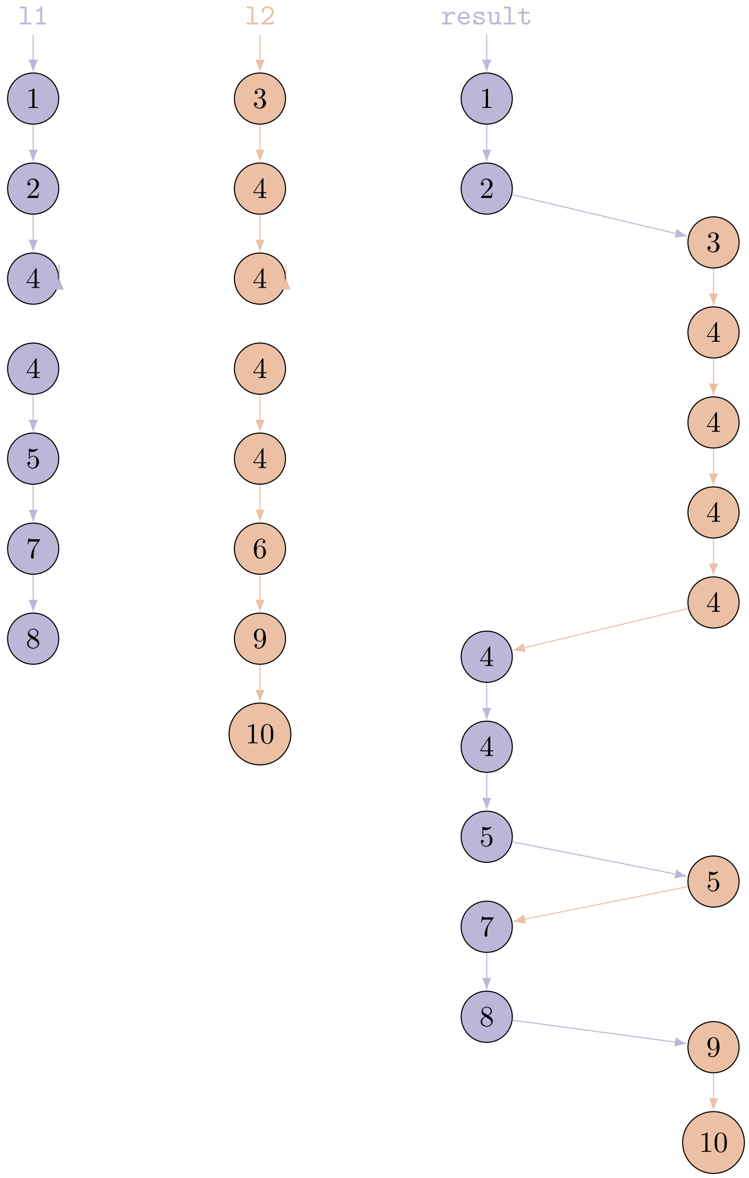
2 In-place mergesort of linked lists of strings

We will now manipulate lists of character strings , in other words objects of the class `Singly<String>`.

2.1 Principle of the in-place mergesort

Merging (`merge`) two ordered linked lists `l1` and `l2` in place is done without creating or destroying any cells. As the following Figure 2 suggests, the algorithm only modifies, as needed, the pointers that point to the next cells. In particular, lists `l1` and `l2` are destroyed.

Figure 2: in-place mergesort



2.2 Implementation

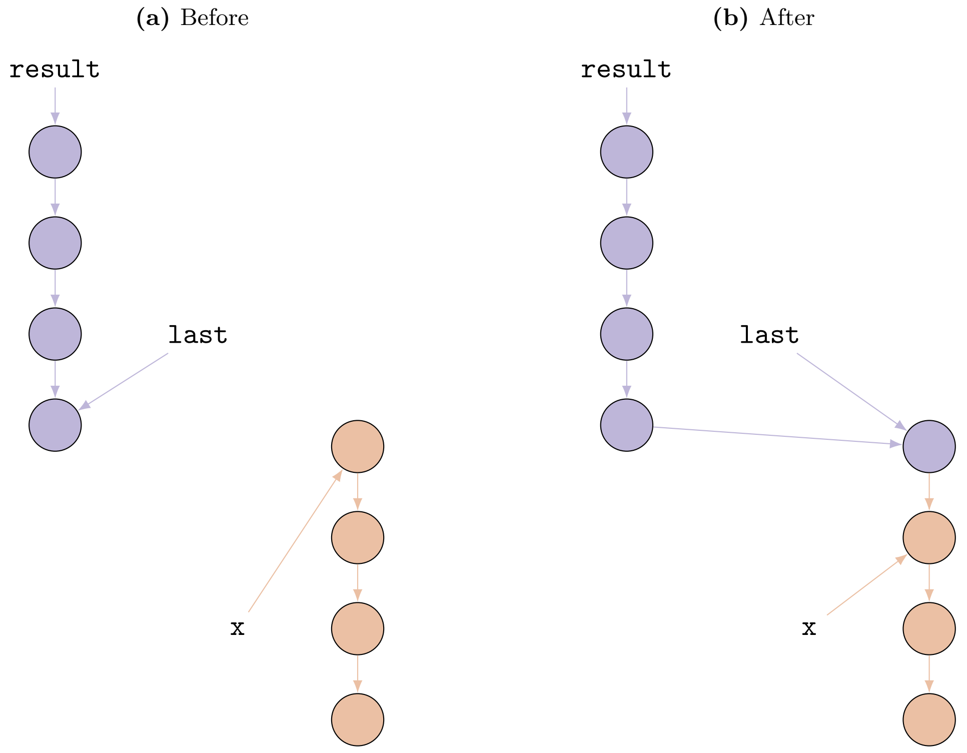
The linked lists `l1` and `l2` are two "reserves" whose elements are "taken" to "put" them in an auxiliary list `result` (i.e. a pointer to the head of the list that will have to be returned). The algorithm is therefore as follows:

- Initialization :
  - *if one of the "reserves" is empty*, the other is returned,
  - otherwise, we initialize a linked list `result` in which we "put" the smallest of the elements `l1.element` and `l2.element`. We also create a variable `last` which, *at the beginning of each iteration*, must point to the last link of the list `result`.

- Iteration :
  - as long as none of the “reserves” are empty, we “take” the smallest of the elements `l1.element` and `l2.element` to “put” it at the end of the list `result`,
  - and as soon as one of the two “reserves” is empty, we stop by returning the concatenation of the `result` list and the other “reserve”.

Hint: “Take” the first element of a list `x` to “put” it at the end of a list `result` whose last link is pointed by `last` is done by modifying four pointers, as suggested in the diagram below. In particular, you must not create new cells, i.e., do not use instructions `new Singly<String>`.

Figure 3


(a) Before **(b)** After

Caution: although logically equivalent, the tests `length(l) == 0` and `l == null` do not have the same algorithmic complexity: make the right choice!

Question 2.2: In the `MergeSortString` class, complete the following static methods:

- `static Singly<String> merge(Singly<String> l1, Singly<String> l2)` performs the merge of two ordered linked lists according to the algorithm described above. In particular, the two lists passed as arguments are destroyed. Remember that the strings must be compared using the `compareTo` method. To avoid causing a stack overflow (i.e. `StackOverflowError`), we will write a loop rather than a recursive method.

- `static Singly <String> sort(Singly <String> l)` performs the merge sort in place of the list passed as an argument. The latter is therefore destroyed. This time, we can easily write a recursive method, because the size of the list is divided by two each time.

Note that this class contains only static methods.

Run the `Test22.java` file which tests the merge and sort methods .

2.3 Application: number of appearances of words in a text

Each object of the `Occurrence` class has the fields:

- `String word`
- `int count`

Question 2.3: In the `Occurrence` class , complete the method

- `static Singly<Occurrence> count(Singly<String> l)`

which returns the list of words present in a list with their multiplicity. We will start by sorting the list passed as an argument, so that identical words are consecutive. No order is specified for the returned list. Test your code by running the file `Test23.java`.

3 Merge sort in place of any linked lists

We can sort the elements of a list as long as they are *totally ordered*, in other words, as long as their class implements the `Comparable` interface .

3.1 Generic implementation

Question 3.1: Adapt the code of the `MergeSortString` class to complete the following methods in the `MergeSort` class :

- `static<E extends Comparable<E>> Singly<E> merge(Singly<E> l1, Singly<E> l2)`
- `static<E extends Comparable<E>> Singly<E> sort(Singly<E> l)`

Run the program contained in the `Test31.java` file to test the generic `merge` and `sort` methods.

3.2 Application: most frequent words

Question 3.2: Using the *generic* `sort` method (which sorts the elements of a list in ascending order) to complete the following methods in the `Occurrence` class :

- `static Singly<Occurrence> sortedCount(Singly<String> l)` which returns the list of words present in the text with their multiplicity so that the most frequent words (i.e. those with the greatest multiplicity) are at the beginning of the list. In addition, with equal multiplicity, we want the words to appear in lexicographical order.

To do this, you must complete, in the `Occurrence` class , the method

- `public int compareTo(Occurrence that)` which returns a strictly negative (resp. positive) value when this is strictly smaller (resp. larger) than `that`, and zero only when `this` and `that` are equal. Here, "small/large" must be interpreted according to the order in which we want the occurrences to be listed.

Caution: The header `class Occurrence implements Comparable<Occurrence>` tells the compiler that objects of the `Occurrence` class are ordered.

Test your code by running the program contained in the `Test32.java` file. The test takes a few seconds to run.

3.3 Application: median value of a list of floats

We now propose to apply our generic sorting in a different context.

A float m is a *median value* of a list of floats when:

- at least half of the elements in the list are *greater than or equal to* `m` and
- at least half of the elements in the list are *less than or equal to* `m`.

We are given the class `Pair.java`.

Question 3.3: In the class `Median`, complete the method

- `static pair<Double> median (Singly<Double> data)`

which returns the range of possible medians as a pair. If the list is empty, the method returns the pair whose two components are equal to the special value `Double.NaN` (an acronym for "Not a Number"). Test your code by running the program contained in the file `Test33.java`.