Notes.  
  
Примечания.

(1) Everywhere in the code, when implementing functional interfaces, use lambda expressions or method references.  
  
(1) Везде в коде, при реализации функциональных интерфейсов, используйте лямбда выражения или ссылки на методы.

(2) Remember that the containers HashSet/ТгееSet do not track potential occurrence of duplicates on changing the state of the objects that these containers store. The same concerns the keys of HashMap/ТгееМар containers.  
  
(2) Помните, контейнеры HashSet/TreeSet не отслеживают потенциальное возникновение дубликатов при изменении состояния объектов, которые хранит этот контейнер. Те же проблемы ключей у HashMap/TreeMap контейнеров.

(3) In the task 6, a ‘word’ should be considered a continuous sequence of Latin letters.  
  
(3) В задании 6, «word» должно считаться непрерывную последовательность латинских букв.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 1

———————————————————————————————-

Package: com.epam.rd.java.basic.practice6.part1

Classes:

Part1 - contains mock of the console input / output and call of the method WordContainer.main  
  
Part1 – содержит макет консольного ввода/ вывода и вызов метода WordContainer.main

WordContainer - contains the required functionality   
  
WordContainer – содержит необходимый функционал.

Word - is set in correspondence to a model ‘a word with information about its frequency in the text’  
  
Word – устанавливается в соответствии «слово с информации о количестве повторений в тексте».

———————————————————————————————-

Develop an application that reads text from console input and displays words in decreasing order of frequency of their occurrence in the text (if the frequencies coincide, the order is lexicographical). A ‘word’ should be considered a continuous sequence of non-whitespace characters.  
  
Разработайте приложение которое читает текст с консоли ввода и отображает слова в порядке убывания частоты их повторений в тексте (Если количество повторений совпадает, используйте лексикографический порядок). «Слово» следует рассматривать как непрерывную последовательность не пробельных символов.

Solve the task using the OOP approach.  
  
Решите задание использую ООП подход.

 The Word class, which contains a string field named ‘content’ and the integer field named ‘frequency’, the WordContainer container aggregates Word objects.  
  
Класс Word, который содержит строковое поле с названием «content» и числовое поле с названием «частота», WordContainer объединяет объекты Word.

When using container core classes, correctly implement (in case of need) the methods Word#equals / Word#hashCode / Word#compareTo.  
  
При использовании классов ядра контейнера, правильно реализуйте (В классах если нужно) методы word#equals / word#hashCode / word#compareTo.

When starting, WordContainer.main reads from the standard input stream (console). Console input generally may contain several lines, each line may contain several words. The word ‘stop’ is a sign of console input end.    
  
При запуске, WordContainer.main читает со стандартного потока ввода (консоли). Консольный ввод может содержать несколько строк, каждая линия может содержать несколько слов. Слово «Stop» обозначает конец консольного ввода.

Demonstrate the functionality of the application (Part1.main).

Input Example

——————————————————

asd 43 asdf asd 43

434 asdf

kasdf asdf stop asdf

stop

——————————————————

Output Example

——————————————————

asdf : 3

43 : 2

asd : 2

434 : 1

kasdf : 1

——————————————————

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 2

———————————————————————————————-

Package: com.epam.rd.java.basic.practice6.part2

Class name: Part2

———————————————————————————————-

Write a program that simulates the following process.

There are n people in the circle, numbered from 0 to n-1.

When counting in a circle, each k-th person (0 <k <n) is crossed out until only one is left.

Example for n=7 and k=3

Initial state:

0123456

Iterations:

013456

01346

0346

034

03

3

Solve the task in two ways:

1)  using deletion of an element by index by the container itself

[List#remove(int)]

——————————————————

// the method returns time of the work

public static long removeByIndex(List<Integer> list, int k) {}

——————————————————

2)  by removing an element using an iterator [Iterator#remove()]

——————————————————

// the method returns time of the work

public static long removeByIterator(List<Integer> list, int k) {}

——————————————————

Compare the run time of each method on the same input data (set in Part2.main) using the ArrayList and LinkedList objects.

As input, use n = 10000, k = 4.

Demonstrate the functionality of the application (Part2.main)

Sample output (stick to the following format).

——————————————————

ArrayList#Index: 13 ms

LinkedList#Index: 93 ms

ArrayList#Iterator: 20 ms

LinkedList#Iterator: 15 ms

——————————————————

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 3

———————————————————————————————-

Package: com.epam.rd.java.basic.practice6.part3

Class names: Part3, Parking

———————————————————————————————-

Implement the Parking class, which simulates the operation of an n-local car parking.

A car drives up to a certain place k (0 <= k <n) and, if it is not occupied, the car takes it.

If the place k is occupied, then it goes to the right until it finds an empty place.

If there is no free place up to the end of the parking, then the free place is searched from the beginning of the parking.

If all places are occupied, then the state of parking does not change (the car leaves).

Implement this functionality in the method:

——————————————————

// The method returns true if it was possible to park the car

boolean arrive(int k)

——————————————————

Additionally, implement a method simulating the departure of a car:

——————————————————

// The method returns true if there was a car in k position

boolean depart(int k)

——————————————————

Methods throw an IllegalArgumentException if k is outside the range [0, n-1].

Also implement a method that prints out the current state of the parking:

——————————————————

void print()

——————————————————

Mark an empty place with 0;  An occupied place should be marked with 1.

Example for n = 4.

Initial state of the parking: 0000

Parking state after the methods are executed (the value returned by the methods is indicated on the right, after comma):

——————————————————

parking.arrive(2)  // 0010, true

parking.arrive(3)  // 0011, true

parking.arrive(2)  // 1011, true

parking.arrive(2)  // 1111, true

parking.arrive(2)  // 1111, false

parking.depart(1)  // 1011, true

parking.depart(1)  // 1011, false

——————————————————

Demonstrate the functionality of the application (Part3.main).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 4

———————————————————————————————-

Package name: com.epam.rd.java.basic.practice6.part4

Class names: Part4, Range

———————————————————————————————-

Write the Range class, which would be a range of numbers [n, m], where n <m.

The class should implement the Iterable interface.

Implement the iterator so that it iterated over the numbers within this range (from the beginning to the end).

Pass an additional parameter ‘reverse’ to the constructor.

The Range class should have two constructors:

——————————————————

public Range(int n, int m) { ... }

public Range(int n, int m, boolean reverse) { ... }

——————————————————

Example:

——————————————————

Range range = new Range(3, 10);

for (Integer el : range) {

  System.out.printf("%d ", el);

 }

System.out.println();

 // result: 3 4 5 6 7 8 9 10

range = new Range(3, 10, true);

for (Integer el : range) {

  System.out.printf("%d ", el);

 }

 System.out.println();

 // result: 10 9 8 7 6 5 4 3

——————————————————

Do not use container classes.

Demonstrate the functionality of the application (Part4.main).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 5

———————————————————————————————-

Class name: com.epam.rd.java.basic.task4.Part5

Class names: Part5, Tree

———————————————————————————————-

Create a generic Tree class that implements a binary search tree data structure.

Do not use container classes.

Mandatory Tree Elements to implement:

——————————————————

public class Tree<E extends Comparable<E>> {

// adds an element to the container

// if the container has an element equal to compareTo of the one being added,

// then no addition occurs and the method returns false

// otherwise, the element gets into the container and the method returns true

// the first added element becomes the root of the tree

// there is no auto balancing in the tree

public boolean add(E element) {...}

//  adds all elements from the array to the container (call in the ‘add’ method loop, see above)

public void add(E[] elements) {…}

// removes the element from the container

// if there is no such element in the container, then it returns false

// otherwise, it deletes the element and returns true

// IMPORTANT! when deleting an element, the tree should not lose the feature of the binary search tree

public boolean remove(E element) {...}

// prints out the tree so that its tree structure can be seen, see the example below

  public void print() {...}

  // nested class, objects of this class make up a tree

private static class Node<E> {…}

 }

——————————————————

The output result of Part5.main (indentation: 2 spaces)

——————————————————

true

false

~~~~~~~

     0

   1

     2

 3

     4

   5

     6

~~~~~~~

true

false

~~~~~~~

    0

   1

     2

 3

     4

   6

——————————————————

Demonstrate the functionality of the application (Part5.main).

When deleting an element, stick to the following algorithm.

1. If the node does not have child nodes, then it’s needed to replace the corresponding pointer with null in its parent.

2. If the node has only one child node, then you need to create a new relationship between the parent of the node to be deleted and its child node.

3. If the node has two child nodes, then you need to 3.1. find the element following it (this element will not have a right descendant)  3.2. place its left descendant in the place of the found element  3.3. replace the deleted node with the found node.

Visual image of the algorithm (in the picture the tree is “turned upside down”): <a href=‘https://neerc.ifmo.ru/wiki/images/thumb/a/ab/Bst\_del1.png/581px-Bst\_del1.png'>algorithm.png</a>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Task 6

———————————————————————————————-

Package: com.epam.rd.java.basic.practice6.part6

Class: Part6 (accepts command line arguments).

The input data should be uploaded from the ‘part6.txt’ file (should be placed in the root of the project)

———————————————————————————————-

Implement a console application (Part6) that analyzes text.

The format of the command line input parameters for the application (the program should understand both short and long options):

——————————————————

-i (or --input) path to the input file;

-t (or --task) the name of the subtask.

——————————————————

There are 4 command line parameters in total, i.e. two pairs <option\_name option\_value>. The order of options with their values is arbitrary.

Examples of command line parameters for launching the application:

——————————————————

-i input.txt -t frequency

--input input.txt --task length

——————————————————

In the Demo.main method, simulate the transfer of command line parameters to the Part6.main method:

——————————————————

System.out.println("~~~~~~~~~~~~Part6");

Part6.main(new String[] {"--input", "part6.txt", "--task",

"frequency"});

Part6.main(new String[] {"--input", "part6.txt", "--task", "length"});

Part6.main(new String[] {"--input", "part6.txt", "--task",

"duplicates"});

——————————————————

There are three subtasks in total: frequency, length, duplicates.

Subtask Part 1 (frequency)

———————————————————————————————-

Class name: com.epam.rd.java.basic.practice6.Part61———————————————————————————————-

In the input file, find the three words that occur most often (if the frequencies coincide, choose that that occurs earlier), and print them sorted alphabetically in reverse order, in the format: word ==> frequency

Output example:

——————————————————

panda ==> 15

ezhik ==> 20

apple ==> 19

——————————————————

Subtask Part 2 (length)

———————————————————————————————-

Class name: com.epam.rd.java.basic.practice6.part6.Part62———————————————————————————————-

Find the three longest words in the input file and print them out in the format: word ==> the quantity of the letters in the word. The list should be sorted in descending order of the letters quantity in the word.

If two words have the same quantity of the letters, then the word that occurred in the source file earlier should also be earlier in the resulting list.

Output example:

——————————————————

anesthetist ==> 11

kitchen ==> 7

bird ==> 4

——————————————————

Subtask Part 3 (duplicates)

———————————————————————————————-

Class name: com.epam.rd.java.basic.practice6.part6.Part63———————————————————————————————-

In the input file, find the first three words that have duplicates, and print their inversion in upper case.

Output example ——————————————————

ADNAP

TAC

ENIGREBUA

——————————————————