LW 1A assessment points distribution

LW1A task. Compulsory part - 7 points

LW1A task. Optional part - 3 points

LW1A Task

Compulsory part (7 points):

1. Download the corresponding IDE Project Folder for the installed IDE (IntelliJ or NetBeans) and make sure it runs. **(1 point)**

2. Implement the unimplemented methods (Implement LinkedList operations (Methods)) for the project LW1_Intro and test them. **(2 points)**

3. Create an interface stack <E> that has the following abstract methods:
   a. `E pop()` – to delete and return the first item in the stack.
   b. `void push(E item)` – to insert a new element at the beginning of the stack.
   c. `E peak()` – for returning the first element in the stack.
   d. `boolean isEmpty()` –to check if stack is empty.

4. Create two classes: `ArrayStack<E>` and `LinkedListStack<E>`. Both classes must implement the `Stack<E>` interface. `ArrayStack<E>` implements stack data structure based on array (array implementation **can use Java collection class** `ArrayList<E>`), `LinkedListStack<E>` implements stack data structure based on linked list (implementation can use Lab1b `LinkedList` class or Java collection class `LinkedList<E>`). Test the operations of these data structures. **(2 points)**

5. Create a `Queue<E>`, interface that has the following abstract methods:
   a. `void enqueue(E item)` – to add a new item to the end of the queue.
   b. `E dequeue()` – to delete and return the first item in the queue.
   c. `E peak()` – to return the first item in the queue.
   d. `boolean isEmpty()` – to check if the queue is empty.

6. Create two classes: `ArrayQueue<E>` and `LinkedListQueue<E>`. Both of these classes must implement the `Queue<E>` interface. `ArrayQueue<E>` implements a queue data structure based on an array (the arrayList system collection class `ArrayList<E>` can be used to implement the array), `LinkedListQueue<E>` implements a queue data structure based on a linked list (for implementation you can use `Lab1b` generic `LinkedList` class or `LinkedList<E>` Java collection class). Test the operations of these data structures. **(2 points)**

Optional part (3 points)

By Use of stack or queue data structures you create and solve the following tasks:

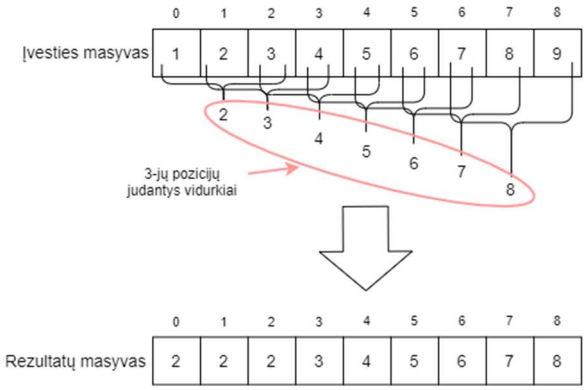
1. **Task 1 (1 point).** We have a string of characters of length N (0 < N <= 1000) consisting of the following sequence of characters: "{", "(", "[", "]", ")" and "}". The string of characters must be balanced, each consecutive parenthesis character "{", "(" or "[" must have the corresponding parenthesis character "}", ")" or "]". Write an algorithm that can verify that the string of characters of length N entered is balanced - that each character that opens the parentheses has a matching character that closes the parentheses. There can be no unbalanced parentheses of another type inside one type of parentheses (for example, the string "{(})" is unbalanced because there is an unbalanced sequence between "{}" and "()").

   The asymptotic complexity of the algorithm should not exceed O(n).

   Test your algorithm with given example.

Algorithm input	Algorithm output
“}”	False
“{()}{[]}”	True
“[{}}”	False
“{()[{}]}{}”	True
“{(})”	False
„([(]{)})“	False

1. **Task 2 (1 point).** You should implement a moving average filter of length k (1 <k <= 10) (shifting right for every single element execution). The input consists of a sequence of real numbers of length N (k <= N <= 100000). The following is an example of a calculation where k = 3:



   Note that the length of the result sequence shouldn’t differ from the length of the input sequence. If this is the case, where it will be so, the result of first k calculation should be equal to the missing first indices of the output. The asymptotic complexity of the algorithm should not exceed O (kn).

   Test your algorithm with given values

Algorithm input	Algorithm output
k = 3 [1 2 3 4 5 6 7 8 9]	[2.0 2.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0]
k = 4 [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]	[2.5 2.5 2.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5 12.5 13.5 14.5]
k = 10	[5.5 5.5.5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.5]

[1 2 3 4 5 6 7 8 9 10]	
k = 2 [5 6 9 1 5 11 3]	[5.5 5.5 7.5 5.0 3.0 8.0 7.0]

2. **Task 3 (1 point).** The printer management system accepts jobs to be printed from users. Users send queries to the printer that consist of the document name and the number of pages. The printer management system wants to distribute jobs evenly and without violating the queuing principle, which means that the earliest documents will be processed first. However, the printer has a limitation: when it receives a single document request, it prints only one page and returns the unfinished document to the end of the document queue. It then takes another request and performs the same procedure. When the printer finishes printing all pages of the document, the document is sent to the job queue.

   Task: Simulate the operation of the printer control system and print the order in which the document sheets will be printed.

   Test your algorithm with given example.

Queue of documents to be printed	Queue of work performed
document1, 3pg document2, 1pg document3, 2pg document4, 1pg	document1, 1/3pg document2, 1/1pg document3, 1/2pg document 4, 1/1pg document1, 2/3pg document3, 2/2pg document1, 3/3pg
dokumentas1, 1psl dokumentas2, 1psl dokumentas3, 1psl	document1, 1/1pg document2, 1/1pg document3, 1/1pg

LW1A Defense

During the defense, teachers can ask students to complete the tasks below and answer the questions below. The list of questions and tasks is not exhaustive - teachers can ask questions related to the topic or practical tasks that are not on this list during the defense.

Possible practical tasks from the compulsory part:

Implement one or more of the following methods:

Return type	**Method** (Description)
void	**add(int index, E element)**
	Inserts the specified element at the specified position in this list.
boolean	**addAll(LinkedList<? extends E> c)**
	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll(int index, LinkedList<? extends E> c)**
	Inserts all of the elements in the specified collection into this list at the specified position.
boolean	**contains(Object o)**

	Returns true if this list contains the specified element.
boolean	**containsAll(LinkedList<?> c)**
	Returns true if this list contains all of the elements of the specified collection.
boolean	**equals(Object o)**
	Compares the specified object with this list for equality.
int	**indexOf(Object o)**
	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	**lastIndexOf(Object o)**
	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove(int index)**
	Removes the element at the specified position in this list.
boolean	**remove(Object o)**
	Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll(LinkedList<?> c)**
	Removes from this list all of its elements that are contained in the specified collection.
boolean	**retainAll(LinkedList<?> c)**
	Retains only the elements in this list that are contained in the specified collection.
E	**set(int index, E element)**
	Replaces the element at the specified position in this list with the specified element.
List<E>	**subList(int fromIndex, int toIndex)**
	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
void	**addFirst(E e)**
	Inserts the specified element at the beginning of this list.
void	**addLast(E e)**
	Appends the specified element to the end of this list.
E	**removeFirst()**
	Removes and returns the first element from this list.
boolean	**removeFirstOccurrence(Object o)**
	Removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
E	**removeLast()**
	Removes and returns the last element from this list.
boolean	**removeLastOccurrence(Object o)**
	Removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
void	**removeRange(int fromIndex, int toIndex)**
	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.

Possible practical tasks from the optional part

1. Determine the maximum depth of the bracket pairs (Task 1)?
2. Find how many different pairs of parentheses (Task 1)?
3. Replace the moving average filter with the median, minimum, or maximum filters (Task 2).
4. Is it possible to solve Task 2 faster than O (kn)? If so, how?
5. In Task 3, the format of the request's changes. Queries that have a priority attribute appear. Document sheets for such requests are printed immediately. Simulate the behavior by adding queries with a priority attribute.

Possible theoretical questions

1. What is the asymptotic complexity of various operations performed in laboratory work?
2. What are the advantages and disadvantages of stack and queue implementations based on an array or linked list?
3. Be able to explain the various nuances associated with software.