Programming Hand in 2

R. Brooks

For this and the remaining handins you must work in your semester project group.

The objective of this hand in is to make you acquainted with the implementations of some of the important data structures that were covered during Lecture 8 and 9:

- Stacks (Program 1 + 2)
- Queues (Program 3)
- Binary Search Trees (Program 4)

Note, in the Java standard library the class `Deque<T>` can be used as stacks and queues (see the link below to Aarhus University's excellent guide to Java's standard library for data structures).

For each of the below exercises there is an associated template. You can find all templates in the zip file 'dmaProg2.zip'. Create a new project in IntelliJ for all your DMA programming handins. It is up to you how you want to structure your hand ins but it may be a good idea to have a module for each of the four hand ins. Unpack the zip file to the newly created project folder.

Do not change any of the existing code, e.g. do not rename methods, etc., although you may need to change the reference to the package (first line in templates). You will only need to add the body of your methods in the relevant template. For each of the programs, replace the comment with your code. Note, you do not need to create methods for user inputs. All you really need to do is to write the logic of the methods, i.e. the algorithms.

Each exercise is accompanied by a test class. You can use this to test whether your method works correctly. Note, only in the first exercise do you get extra points for a fast solution. You must analyze your code yourself and comment on the complexity. Comments are made in the code as per the example shown in class and which is similar to what you must do in your SEP1 project.

For this and the final programming hand ins, it will be useful to get acquainted with data structures in Java's standard library. You've already been introduced to ArrayLists. Aarhus University has made a nice guide to which libraries you need. You can find the guide here. Some of the below exercises are also from Aarhus University.

To find a fast solution to Program 4, you will need to use a Balanced binary search tree (BBST), so you can use a `TreeSet` in Java. In class, we went through AVL trees as an example of a BBST. Note, the link mentions a red/black search tree which is simply another kind of BBST but which has "the same" implementation in Java (i.e. `TreeSet`).

The scoring is stated in each exercise but note that algorithm analysis gives you extra points. Naturally, you only need to analyse the code you have written so you do not need to comment on existing code in the templates. You can at most obtain 9 points in this hand in which means you need 5 points to get it approved.

You upload your hand in as a zip-file where you simply zip the handin2 module with all your code/files.

If you get stuck doing any of the exercises, it is recommended seeking out the student instructor at the study café.

Program 1: `Reverse Polish Notation`

When writing an arithmetic formula such as 1 - (2 * 3) in Reverse Polish notation (RPN), we write the operator, e.g. minus, after the operands. To multiply 2 by 3, we write `2 3 *` instead of `2 * 3`.

The formula `1 - (2 * 3)` becomes `1 (2 3 *) -`, and we may remove the parentheses, resulting in `1 2 3 * -`, since there is no ambiguity about where they should be placed.

A formula written in RPN can be evaluated using a calculator and a stack in the following way: Read the formula from left to right.

- When you encounter a number, push it on the stack.
- When you encounter an operator, remove the two top elements on the stack, compute the operation (plus, minus, times), and push the result on the stack.

For example, consider the expression

```
(20 − (2 ∗ (3 + 5))) ∗ (2 − 4)
```

which, when written in RPN, becomes

```
20 2 3 5 + ∗ − 2 4 − ∗
```

When you read the 11 symbols from left to right, the stack changes as follows:

- Push: 20
- Push: 20 2
- Push: 20 2 3
- Push: 20 2 3 5
- Plus: 20 2 8
- Times: 20 16
- Minus: 4
- Push: 4 2
- Push: 4 2 4
- Minus: 4 -2
- Times: -8

In this exercise you must implement a class named `ReversePolishCalculator` with the following methods:

- `void push(int n)` $-\!>$ pushes n on the stack
- `void plus()` $-\!>$ pops the two top elements from the stack and pushes their sum
- `void minus()` $-\!>$ pops the two top elements from the stack and pushes their difference
- `void times()` $-\!>$ pops the two top elements from the stack and pushes their product
- `int read()` $-\!>$ returns the top element from the stack (without removing it)

Your implementation is allowed to throw an exception if a method is called at a wrong time, for instance if `read()` is called when the stack is empty or if `plus()` is called when the stack has fewer than two elements.

Note that for minus, the top element on the stack should be subtracted from the second-from-top element. In the previous example, the top of stack was the rightmost element.

Concretely, you should use the template **ReversePolishCalculator.java**, and implement the methods `push, plus, minus, times`, and `read`. You are allowed to add your own private fields to the class.

Scoring:

- 1 point for correct algorithm and 1 point for correct algorithm analysis

Program 2: `Balanced parenthesis checking`

Have you ever forgotten to end a parentheses? Well those days are over because now you are going to program your own parenthesis checker. In this problem, the input to your program is an array of N characters, each of which is either `(`, `)`, `[` or `]`, and your program must return `true` if the parentheses are properly matched, and `false` if they are not.

For example, the following arrays of parentheses are properly matched:

```
( ( ) ( ) ) [ ]
[ ( ( ( ) ) [ ] ) ]
[ [ ] ] ( )
[ [ ] [ [ ] ] ]
( ) [ ( [ ] ) ] [ ]
```

The following array is NOT properly matched since the round parentheses are not aligned:

```
( [ ] (
```

The following array is NOT properly matched since the square parentheses do not face each other:

```
( ] [ )
```

The following array is NOT properly matched since the round parenthesis matches a square parenthesis:

```
]
    ( [ ] ]
```

The following array is NOT properly matched since the round parenthesis intersect the square parentheses:

```
( [ ) ]
```

Concretely, you should use the template **parenthesis.java** and implement the method `checkParentheses` which takes an `ArrayList<Character>` input and returns a `boolean`.

You may assume that the input only contains the characters `'('`, `')'`, `'['`, `']'`, so you do not have to check if there are any other kinds of characters in the input.

Scoring:

- 1 point for correct algorithm and 1 point for correct algorithm analysis

Constraint: You must use a stack to contain the open parentheses. An $O(N)$ time algorithm is fast enough, and this will be accomplished by using a stack.

If you use the `remove()` method on the input `ArrayList`, your program might take $O(N^2)$ time and not be fast enough.

Program 3: `Queue simulation`

In this problem, the input to your program is an array of N integers, and your program should repeat the following operation as long as there are at least two integers:

- Remove the first two integers from the list and add the second one back to the end of the list.

When there is just one integer left, your program should return it.

For example, if the input is the list

```
1 2 3 4 5
```

then your program should perform the above operation four times:

- After the 1st operation, 1 and 2 are removed and 2 is added, so the list contains: 3 4 5 2
- After the 2nd operation, 3 and 4 are removed and 4 is added, so the list contains: 5 2 4
- After the 3rd operation, 5 and 2 are removed and 2 is added, so the list contains: 4 2
- After the 4th operation, 4 and 2 are removed and 2 is added, so the list contains: 2

Concretely, you must implement a public method named `simulate` that takes an `ArrayList<Integer> input` as argument and returns an `int`. Use the template `QueueSimulation.java`.

Scoring:

- 1 point for correct algorithm and 1 point for correct algorithm analysis

Constraint: An $O(N)$ time algorithm is required, so your program should use a queue data structure. If you only use an ArrayList, your program might take $O(N^2)$ time and not meet the constraint.

Program 4: `dodgeBall`

Crazy Dodgeball is a game in which two teams play against each other using a ball. At any point in time, the players on your team stand on a line, and the following two things can happen:

- A new player comes alive and is added to the line at position x
- A ball is thrown at position x
  - If a ball is thrown at position x and there is a player at position x, that player is out and is removed from the line.
  - Otherwise, the player that is closest to position x moves to position x and throws the ball back at the other team.

Here is an example, starting with four players at positions 1, 7, 10 and 20:

- Ball thrown at 10: 1, 7, 20 (10 dead; distance 0)
- Ball thrown at 18: 1, 7, 18 (moved from 20 to 18; distance 2)
- Ball thrown at 5: 1, 5, 18 (moved from 7 to 5; distance 2)
- New player at 11: 1, 5, 11, 18
- Ball thrown at 8: 1, 8, 11, 18 (moved from 5 to 8; distance 3)

This example corresponds to `test1` in the `testDodge` class. Looking at how the test case is constructed may sometimes be useful in figuring out how to construct the code (this is similar to "test-driven" development).

In this exercise you must implement a class named `dodgeBall` with the following methods:

- `void addPlayer(int x)` $\longrightarrow$ adds a new player at position x in the game
- `int throwBall(int x)` $\longrightarrow$ updates the set of players when a ball is thrown at position x and returns the distance from the closest player to the ball

At no point in time are there two players that stand at the same position. If a ball is thrown at position x and there are two players that are both closest to the ball, that is, they stand at positions $x - d$ and $x + d$ for some integer d, then the player with the numerically smallest position moves to the ball.

For example, if there are two players that stand at positions 1 and 7, and a ball is thrown at position 4, then the player at position 1 moves to position 4.

Concretely, you should use the template **dodgeBall.java** and implement the methods `addPlayer` and throwBall in `dodgeBall`, and you are allowed to add your own private fields to the class.

Input constraints:

- When `throwBall(x)` is called, there is at least one player on the line
- Number of operations ≤ 500000
- $1 \leq x \leq 5000000$ (whenever `addPlayer(x)` or `throwBall(x)` is invoked)

Scoring:

- 1 point for correct algorithm
- 1 extra point for correct and fast algorithm
- 1 additional point for correct algorithm analysis

An algorithm that spends $O(\log N)$ time on `addPlayer()` and `throwBall()` is fast enough for the extra point.

Hint for the fast solution: Use a balanced binary search tree.