Algorithms & Datastructures
Laboratory Exercise Sheet 1

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

March 10, to be discussed on March 16

Exercise 1) Java Environment First of all your Java environment should be set up if you haven't got one yet. You can use any IDE (e.g. Eclipse) or editor. Please keep in mind, the exams will be pen&paper. This exercise is for getting in touch with Java and the Stack.

Stacking

```
public class HelloStack {

   public static void main(String[] argv) {
      System.out.println("Hello Stack");

      Stack stack = new ListStack();
      stack.push(1);
      stack.push(2);
      stack.push(4);
      stack.push(8);
      stack.push(16);

      for(int i = 0; i<5; i++) {
         int item = stack.pop();
         System.out.println(item);
      }
   }
}
```

1. Using the list-stack from the lecture, compile and run the above code. What is the result?
2. How many operations need to be done?
3. What happens, if you add some `push` or `pop`?
4. Optimize the method `main` for operation counts. (Resulting to the same output.) **(Send in the code!)**

Exercise 1) Solution

1. 16\n8\n4\n2\n1
2. The ListStack has for each method 4 operations (lecture)
   Stack: 20 + 20
   Rest: 7+5
   but depends on measurement
3. Change in output, but only 5 elements are outputed
   Runtime-Errors if you add more pop than pushes
4. One possibility: Change ListStack to ArrayStack
   Second one: Eliminate the stack completly.

Eliminate Stack

```
public class HelloStackAlternative {
   public static void main(String[] argv) {
      System.out.println("Hello Stack");
      System.out.println(16);
      System.out.println(8);
      System.out.println(4);
      System.out.println(2);
      System.out.println(1);
   }
}
```

Exercise 2) Stack Usages Stacks are not appropriate for everything, however, they are used quite extensible in programming. Actually, they are used in *every* program you can imagine, even in the most simple "Hello World" one as the compiler generates at least one. On some architectures, there are even primitives at the assembler/machine-code level.

Parenthesis

```
public class Parenthesis {

   public static void main(String[] argv) {
      if (argv.length<1) {
         System.err.println("Need a string as parameter!");
         System.exit(-1);
      }
      String str = argv[0];
      Stack stack = new ListStack();
      boolean error = false;
      int i = 0;
      while (!error && i<str.length()) {
         /* ... */ //str.charAt(i); //need code here
         i++;
      }

      if (true) { //need to be changed
         System.out.println("Parenthesis BAD!");
      } else {
         System.out.println("Parenthesis OK!");
      }
   }

}
```

1. Enhance the above program, so that it does a paranthesis-check. This means, for each "(" there is a corresponding ")", and for each "{" there is a corresponding "}". All other characters are ignored. Both paranthesis are not overlapping (so, the following is incorrect: "(({)})"). **(Send in the code!)**
   Note: You will need some additional method in the Stack.

2. Run your program with diverse input strings and test it.

3. How many stack-operations are used for a string with 10 (paranthesis-only) characters? How many with 1000 chars?

4. Imagine an stack-implementation with `push` having 1 operation and `pop` having 8 ones. Would this have performed better?

5. Would an extension for "[]" and other paranthesis have changed the performance behavior or the code-bloat? What do you conclude?

Exercise 2) Solution

1. Note: The switch could even be split into two functions. But as the students would've problems with that, I didn't do that.

Parenthesis

```
public class ParenthesisSolution {

   public static void main(String [] argv) {
      if (argv.length<1) {
         System.err.println("Need a string as parameter!");
         System.exit(-1);
      }
      String str = argv[0];
      Stack stack = new ListStack();
      boolean error = false;
      int i = 0;
      while (!error && i<str.length()) {
         int corresponding;
         switch (str.charAt(i)) {
         case '(':
            stack.push('(');
            break;
         case '{':
            stack.push('{');
            break;
         case ')':
            if (stack.isEmpty()) {
               error = true;
               break;
            }
            corresponding = stack.pop();
            if (corresponding != '(') {
               error = true;
               break;
            }
            break;
         case '}':
            if (stack.isEmpty()) {
               error = true;
               break;
            }
            corresponding = stack.pop();
            if (corresponding != '{') {
               error = true;
               break;
            }
            break;
         }

         i++;
      }

      if (error || !stack.isEmpty()) {
         System.out.println("Parenthesis BAD!");
      } else {
          System.out.println("Parenthesis OK!");
      }
   }
}
```

Stack

```
interface Stack {
    void push(int n);
    int pop();
    boolean isEmpty();
}
```

ListStack

```
class ListStack implements Stack {
    Node head = null; // head of stack
    public void push(int n) {
        this.head = new Node(n, this.head);
    }
    public int pop() {
        if (this.head == null) {
            throw new RuntimeException("no element");
        } else {
            int n = this.head.value;
            this.head = this.head.next;
            return n;
        }
    }
    public boolean isEmpty() {
        return head == null;
    }
}
```

2. -

3. 20, 2000
   (or 80 and 8000 operations if they multiply by operations done **in** the stack)

4. If the result is (more probable) “OK”: No.
   As then you have got as many `pop` as `push` operations.
   if the result is very probable “NO”, it might be faster.

5. Probably no, it is only constant multiplication. But it depends on your implementation of course.

Exercise Environment

- The exercise sheets are given on wednesday.
- Send in your Java programs til Monday, 10 am.
- They are talked at tuesday.
- Please solve the sheets in working groups of 2 or 3 students.
- Please send in the Java programs to your lab lecturer by email.

Marking/Exam Environment

- For details please refer to the rules-document.
- There will be two exams, one after the first third and one after the second third of the semester. Additionally, you will get a mark for attending and presentating at the lab.
- The exams and the collaboration in lab will be each 33 % of your resulting mark.
- You will get a mark (1-5) if you attend to the first exam.
- The exams are closed-book, pen&paper (no computers, no Eclipse!).