Data structures, exercise 3, 9-12 February

1. Familiarize yourself with the Java API classes ArrayList and LinkedList. Implement a queue with the help of both of them.

   Compare the performance of your queue implementations empirically, i.e. measure how well your queues function with large inputs of different length. One way to measure the time spent on the performance is:

   ```
   int times = 100000;
   ArrayListQueue j1 = new ArrayListQueue();
   long begin = System.currentTimeMillis();
   for ( int i=0; i<times; i++ ) j1.enqueue(i);
   for ( int i=0; i<times; i++ ) j1.dequeue();
   long end = System.currentTimeMillis();
   System.out.println( (end-begin)+" milliseconds");
   ```

   Draw a figure of the time used by each version in relation to the number of performed operations.

   What conclusions can you draw from an empirical analysis of the queue performances? How does an empirical analysis relate to an $\mathcal{O}$-analysis? Why is there a difference in performance?

2. Implement (with Java or some other programming language) a stack based on an array. Increase the size of the array if there is no place for a new element in the array when a push operation is made.

   Implement two versions of the stack: one where the size of the array is always doubled, and another where the size always increases with 100. Both arrays start out with the size 100.

   Compare the performance of your implementations empirically. Again, we are most interested in large inputs. Draw an figure of the time spent by each version in relation to the number of performed push operations. What conclusions can you draw from an empirical analysis of the stack performances? Why is there a difference in performance?

3. Use Java (or some other programming language) to implement a singlylinked list, in which integers are stored. In singly-linked lists, the nodes do not have prev attributes. You can find a singly-linked list in the figure of problem 6.

   The list contains the following methods:

   - `delete(k)` removes the integer k from the list (so this is not quite the same as the operation in the Cormen, where the parameter was a reference to the list node to be removed)
   - `search(k)` tells whether the integer k is listed
   - `add(k)` adds the integer k to the list
   - `list()` prints the integers on the list

   Write a main program, as well, to test the list. What is the time requirement of the methods?

4. Change your implementation so that the list stores all its keys in increasing order. How does the time complexity of the methods change? **Bonus:** Implement the method `reverse()` for your list to reverse the contents of the list; i.e. the first list element of the list becomes the last, etc. The method should have space requirement $\mathcal{O}(1)$. It can use $\mathcal{O}(n)$ time, where n is the number of elements in the list.

   You do not have to complete this bonus question to gain your exercise points.

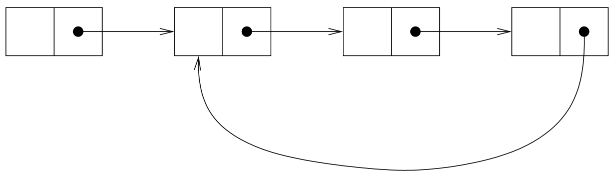
5. A linked list can be represents as an array, where links to the nodes are simply indices of the array. For example, the (unordered doubly-linked) list $(68, 24, 15, 17)$ might have an array representation

	key	*next*	*prev*
1:			
2:			
3:	24	7	5
4:			
5:	68	3	0
6:			
7:	15	8	3
8:	17	0	7
9:			
10:			

head: 5

Show how to implement INSERT ja DELETE using such an array representation. You solution should be such that an array with n rows is sufficient if the list never contains more than n elements. In other words, the INSERT procedure must be able to re-use the rows released earlier by DELETE.

6. Due to a programming error, we may end up with a loop in a list structure:



Give an algorithm to test whether a given list contains such a loop. Your algorithm should leave the list unchanged. How much time and space does your algorithm take?

Additional challenge: algorithm uses only constant amount of space.