1 Assignment No. 4: Merge *k* Ordered Lists Efficiently

Allocated time: 2 hours

1.1 Implementation

You are required to implement **correctly** and **efficiently** an O(nlogk) method for **merging *k* sorted sequences**, where n is the total number of elements. (Hint: use a heap, see *Seminar no. 2* notes).

Implementation requirements:

- Use linked lists to represent the k sorted sequences and the output sequence

Input: k lists of numbers $< a_1^i, a_2^i, \ldots, a_{m_i}^i >$, $\sum_{i=1}^{k} m_i = n$
Output: a permutation of the union of the input: $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

1.2 Minimal requirements for grading

The lack of any of the minimum requirements (even partially) may result in a lower grade through penalties or refusal to accept the assignment resulting in a grade of 0.

- *Demo*: Prepare a demonstration of correctness for each algorithm implemented. The correctness of each algorithm is demonstrated through a simple example (maximum 10 values).
- The charts created must be easy to evaluate as in grouped and added through the Profiler functions as specified by the assignment requirements. The assignment will not be evaluated if it contains a plethora of ungrouped charts. For example, the comparative analysis implies the grouping of the compared algorithms.
- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- *The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and* ***the correct answer to the questions from the teacher.***

1.3 Requirements

1.3.1 Generate *k* randomly sized sorted lists (having *n* elements in total, *n* and *k* given as parameters) and the merging of 2 lists (5p)

Demo: You will have to show your algorithm (*generation and merging*) works on a small-sized input (e.g. k=4, n=20).

1.3.2 Adapt *min-heap* operations to work on the new structure and the merging of k lists (3p)

Demo: You will have to show your algorithm (*merging*) works on a small-sized input (e.g. k=4, n=20).

1.3.3 Evaluation of the algorithm in average case (2p)

We will make the **average case** analysis of the algorithm. Remember that, in the average case, you have to repeat the measurements several times. Since both **k** and **n** may vary, we will make each analysis in turn:

- Choose, in turn, 3 constant values for *k* (**k1=5, k2=10, k3=100**); generate *k* **random** sorted lists for each value of *k* so that the number of elements in all the lists *n* varies between **100 and 10000**, with a maximum increment of 400 (we suggest 100); run the algorithm for all values of *n* (for each value of *k*); generate a chart that represents the **sum of assignments and comparisons** done by the merging algorithm for each value of *k* as a curve (total 3 curves).
- Set ***n = 10.000***; the value of *k* must vary between 10 and 500 with an increment of 10; generate *k* **random** sorted lists for each value of *k* so that the number of elements in all the lists is 10000; test the merging algorithm for each value of *k* and generate a chart that represents the **sum of assignments and comparisons** as a curve.