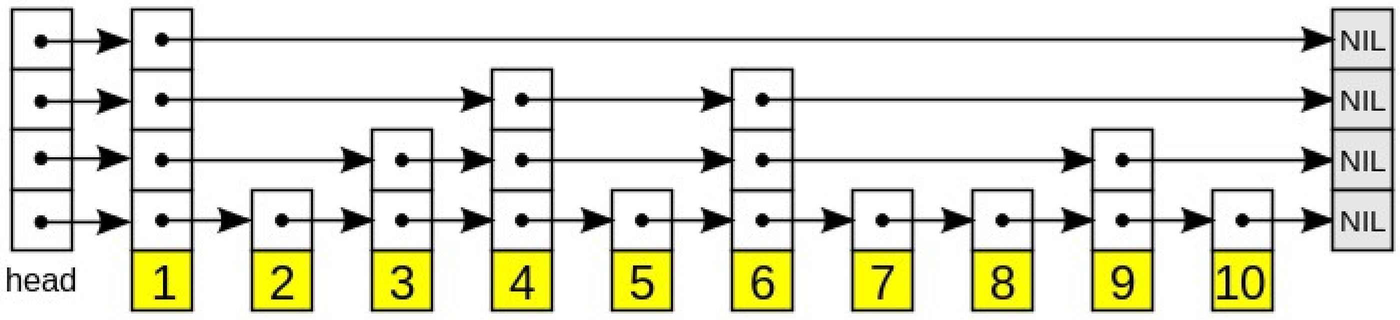
Freestyle Assignment 20P, Manuel Haag 2243192

submission date: 13.02.2022 (23:59)

5.1 Skiplist (13P)



In this exercise you should construct a *concurrent Skiplist* (https://en.wikipedia.org/wiki/Skip_list).
It supports the following operations:

- *add*(e): $X := X \cup \{e\}$
- *contains*(e): $e \in X$
- *remove*(e): $X := X \setminus \{e\}$

a) Sequential Implementation (1P)

Start by implementing a sequential Skiplist. *[1]*

b) Concurrent LazySkipList using Locks (4P = 3P + 1P)

Implement a concurrent *LazySkipList [2]* supporting the same operations as the sequential one using locks.

- First, implement a concurrent *LazySkipList* supporting *add* and *contains*. *Add* returns if the operation was successful and *contains* should be implemented wait-free, i.e. no thread waits during the operation **(3P)**.
- Now implement the *remove* operation. The operation also returns wether the operation was successful. **(1P)**

c) Lockless Variant (4P = 3P + 1P)

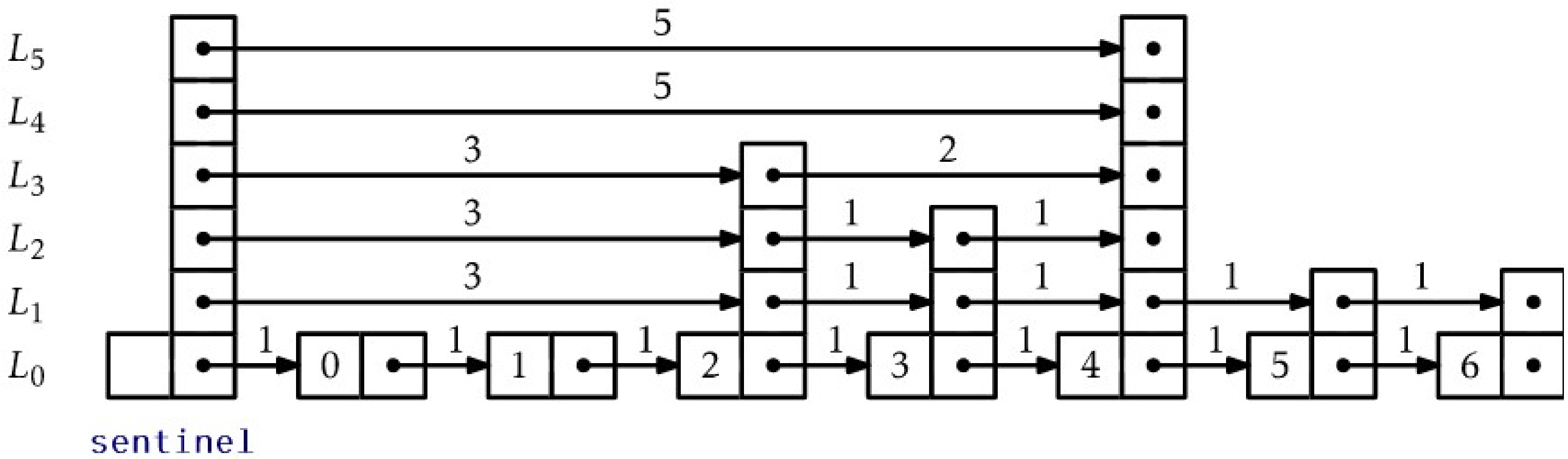
Now implement a lockless variant *[3]*. Can the *Skiplist* be kept consistent? Implement a lockless variant that is as good as possible.

- Begin by implementing a variant supporting *add* and *contains* **(3P)**.
- Now implement the *remove* operation **(1P)**.

d) Evaluation (4P)

Create a benchmark to test your *Skiplist* with varying number of threads.
Try different input patterns (sorted, permuted numbers) and different percentages of operations (*add, remove, contains*).
Vary the parameter p which defines the probabilty of a newly inserted node to increase it's height by one.
Present several reasonable performance measures like time and speedup.
Additionally measure at least one non-standard metric (not running time based).

5.2 Indexable Skiplist (7P)



a) Sequential Indexable Skiplist (1P)

Now make your sequential implementation indexable by storing for each pointer the length of the shortcut it provides. This should allow to access the k-th element.

b) Concurrent Indexable Skiplist (2P)

Adapt this technique for your parallel (lock or lockless) implementation.

c) Rank Operation (1P)

Using this extension implement a concurrent operation *rank(e)* which returns the index of the element e, if e is in the *Skiplist*.

d) Evaluation (3P)

Compare the performance of the concurrent indexable *Skiplist* with your *Skiplist* from b) and c).
Also write a benchmark that test the performance of k-th element and *rank* by comparing it with an alternative algorithm.
E.g. *push_back* and *sort*, for larger intervals of *add* or an other concurrent datastructure.

References

[1] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. ACM Transactions on Database Systems, 33(6):668–676, 1990.

[2] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. Fourteenth Colloquium on structural information and communication complexity (SIROCCO) 2007 pp. 124–138, June 5–8, 2007, Castiglioncello (LI), Italy.

[3] Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[4] The Art of Multiprocessor Programming, Maurice Herlihy, Nir Shavit.
