**Retroactive data structures**

Today's lecture is our second and final lecture on time travel, or more precisely, temporal data structures. Here we will study retroactive data structures, which mimic the "plastic timeline" model of time travel. For example, in *Back to the Future*, Marty goes back in time, makes changes, and returns to the present that resulted from those changes and all the intervening years. In a partially retroactive data structure, the user can go back in time, perform an additional operation, return to the present that results, and query the resulting data structure. In this way, we maintain a single (changing) timeline, consisting of the sequence of update operations. Retroactive data structures can add, or remove, an update at any time, not just the end (present). A fully retroactive data structure can furthermore query the data structure at any time in the past. Finally, a third type of retroactivity, called "non-oblivious", puts queries on the timeline as well, and reports the first query whose answer changed. As you might expect, retroactivity is hard to obtain in general, but good bounds are known for several data structures of interest.

[Details and references]

This lecture introduces the retroactive data structure and a new computation model, the cell probe model. Retroactive data structures maintain a linear timeline and allow updates to be performed at any time [Demaine, Iacono, Langerman -- 2003 / T. Alg 2007]. Partial retroactivity only permits queries at the present time, while full retroactivity allows queries at any time. A third kind of retroactivity, non-oblivious retroactivity, lets the user put a series of queries on the time line, and whenever an update is performed, the data structure reports the first query whose value has changed. For partial and full retroactivity, there are two easy cases. The first one concerns commutative and invertible updates. The second, being slightly more complex, concerns decomposable search problems [Bentley & Saxe -- J. Alg 1980], which is solved using a *segment tree*. The harder situation with general transformation [Demaine, Iacono, Langerman -- 2003 / T. Alg 2007] can be solved naively using rollback. Concerning general transformation's lower bound, it has been proven that $\Omega(r)$ overhead can be necessary [Frandsen, Frandsen, Mitlersen -- I&C 2001]. The same paper also proves a $\Omega((r/\lg r)^{1/2})$ lower bound for the cell probe model. The cell probe model is "a model of computation where the cost of a computation is measured by the total number of memory accesses to a random access memory with log $n$ bits cell size. All other computations are not counted and are considered to be free." [NIST]. A better lower bound for cell probe model, such as $\Omega(r/\text{poly}\lg r)$, is open. With the help of a geometric representation, we show a partially retroactive priority queue with $O(\lg n)$ insert and delete-min [Demaine, Iacono, Langerman -- 2003 / T. Alg 2007]. Other data structures

mentioned include queue, deque, union-find, a fully retroactive priority queue with $O(m^{1/2} \lg m)$ per op and the successor problem [Giora & Kaplan -- T. Alg 2009]. A better fully persistent priority queue is open. The best successor problem solution requires fractional cascading from lecture 3 and the van Emde Boas data structure from lecture 11. For non-oblivious retroactivity [Acar, Blelloch, Tangwongsan -- CMU TR 2007], a priority queue is shown with the help of a geometric representation.

# 6.851 Advanced Data Structures

Spring 2012