List with Move to Front Heuristic

→ in an implementation shown before, we perform access(e) method in time proportional to the index e

→ if e is k^{th} most popular element in list, accessing it takes $O(k)$ time

→ a heuristic (rule of thumb) that takes advantage of the locality of reference that is present in an access sequence is the move-to-front heuristic

- to apply, each time we access an element we move it all the way to the ~~back~~ front with hope that the element will be accessed soon again

Consider a scenario where we have n elements and following n^2 accesses:

- element 1 accessed n times
- element 2 accessed n times
- ...
- element n accessed n times

If we store elements sorted by access counts, inserting each the first time they are accessed,

- each access to element 1 takes $O(1)$
- 2 takes $O(2)$
- ...
- each access to element n runs in $O(n)$ time

Thus, the time is proportional to: $\frac{n \cdot (n+1)}{2} \cdot n$, or $O(n^3)$

Link Based vs. Array Based Sequences

array adv.

→ arrays provide $O(1)$-time access to an element based on an integer index; locating the k^{th} element in a linked list requires $O(k)$ time

→ operations w/ equivalent asymptotic bounds typically run a constant factor more efficiently w/ an array-based structure vs. a linked list structure.

→ array-based implementations usually use less memory proportionally

→ link-based structures provide worst-case time bounds

→ link-based structures support $O(1)$-time insertions & deletions at arbitrary positions

linked list advantages