CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations

Armin Jamshidpey, Éric Schost
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2026

version 2026−06−30 09:52

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests
   - Optimal Static Ordering
   - Dynamic Ordering: MTF

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests
   - Optimal Static Ordering
   - Dynamic Ordering: MTF

Dictionary ADT: Implementations thus far

A *dictionary* is a collection of key-value pairs (KVPs), supporting operations *search*, *insert*, and *delete*.

Realizations we have seen so far:

- **Unordered array or list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Binary search trees**: $\Theta(\mathit{height})$ search, insert and delete
- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete

Improvements/Simplifications?

- **Can show:** If the KVPs were inserted in random order, then the expected height of the binary search tree would be $O(\log n)$.
- How can we use randomization within the data structure to mirror what would happen on random input?

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests
   - Optimal Static Ordering
   - Dynamic Ordering: MTF

Towards skip lists

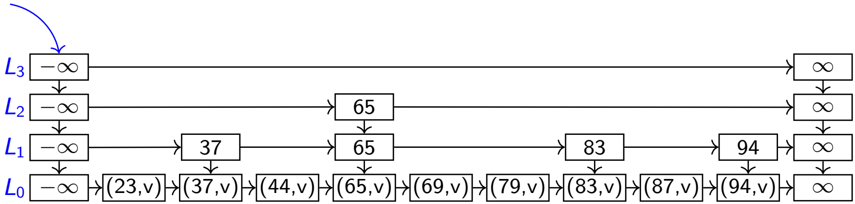
We did not consider an ordered list as realization of ADT Dictionary. Why?

- *insert* and *delete* take $\Theta(1)$ time in an ordered lists, once we know the place where to do them.
- The bottleneck is *search*:
  - In an ordered array, we can do binary search to achieve $O(\log n)$ run-time.
  - In an ordered list, we cannot 'skip to the middle' and so cannot do binary search.
  - Therefore *search* takes $\Theta(n)$ time in an ordered list—too slow.

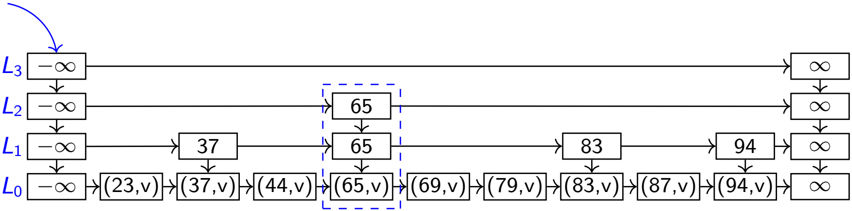
Skip lists

A hierarchy of ordered linked lists (*levels*) $L_0, L_1, \cdots, L_h$:

- Each list L_i contains the special keys $-\infty$ and $+\infty$ (**sentinels**)
- List L_0 contains the KVPs of S in non-decreasing order. (The other lists store only keys and references.)
- Each list is a subsequence of the previous one, i.e., $L_0 \supseteq L_1 \supseteq \cdots \supseteq L_h$
- List L_h contains only the sentinels, all other lists contain at least one non-sentinel.



Skip lists



A few more definitions:

- *node* = entry in one list vs. *KVP* = one non-sentinel entry in L_0
- There are (usually) more *nodes* than *KVPs*
  Here # (non-sentinel) nodes = 14 vs. $n \leftarrow$ # KVPs = 9.
- *root* = topmost left sentinel is the only field of the skip list.
- Each node p has references $p.after$ and $p.below$
- Each key k belongs to a **tower** of nodes
  - Height of tower of k: maximal index i such that $k \in L_i$
  - Height of skip list: maximal index h such that L_h exists

Skip lists: Search

For each list, find **predecessor** (node before where k would be).
This will also be useful for *insert*/*delete*.

get-predecessors (k)

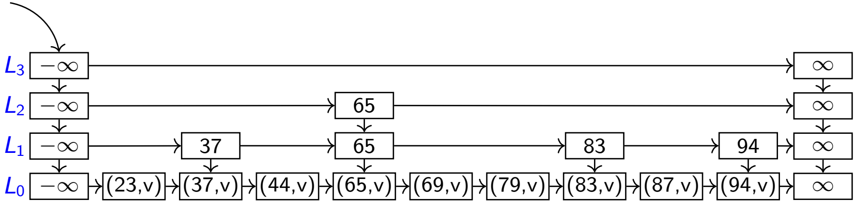
$p \leftarrow$ root
$P \leftarrow$ stack of nodes, initially containing p
while $p.below \neq \texttt{NULL}$ **do**
    $p \leftarrow p.below$
    while $p.after.key < k$ **do** $p \leftarrow p.after$
    $P.push(p)$
return P

skipList::search (k)

$P \leftarrow$ *get-predecessors*(k)
$p_0 \leftarrow P.top()$ // predecessor of k in L_0
if $p_0.after.key = k$ **return** KVP at $p_0.after$
else return "not found, but would be after p_0"

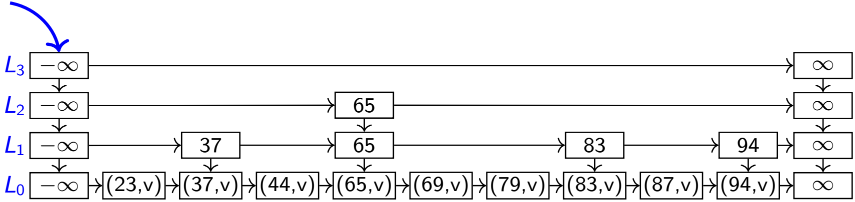
Skip list search: Example

Example: *search*(87)



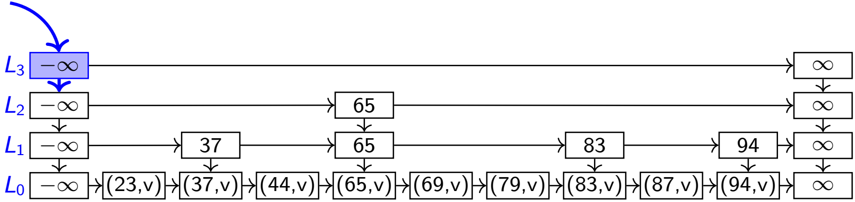
Skip list search: Example

Example: *search*(87)



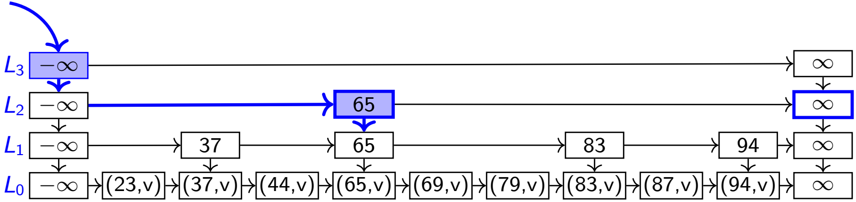
Skip list search: Example

Example: *search*(87)



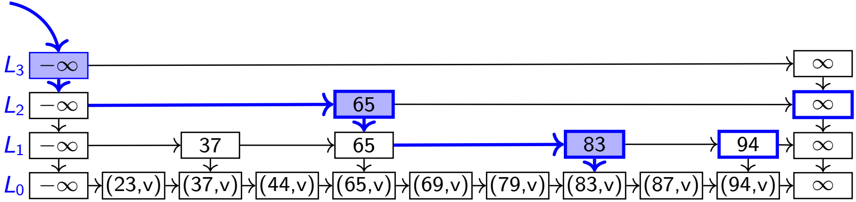
Skip list search: Example

Example: *search*(87)



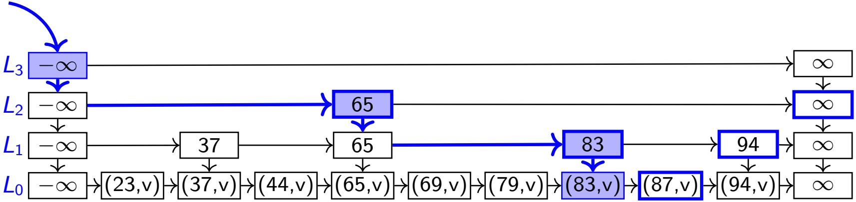
Skip list search: Example

Example: *search*(87)



Skip list search: Example

Example: *search*(87)



key compared with k

added to P

path taken by p

Final stack returned:

(83,v)
83
65
$-\infty$

Skip list: Deletion

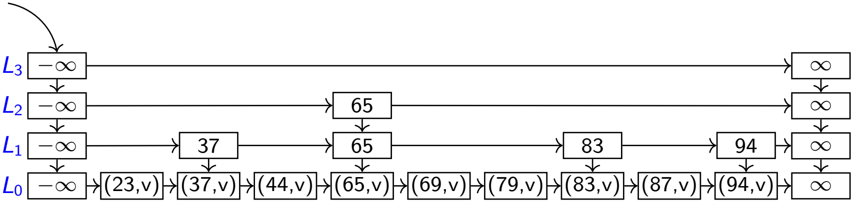
It is easy to remove a key since we can find all predecessors.
Then eliminate lists if there are multiple ones with only sentinels.

skipList::delete(k)

- $P \leftarrow$ *get-predecessors*(k)
- **while** P is non-empty
  - $p \leftarrow P.pop()$ `// predecessor of` k `in some list`
  - **if** $p.after.key = k$
    - $p.after \leftarrow p.after.after$
  - **else break** `// no more copies of` k
- $p \leftarrow$ left sentinel of the root-list
- **while** $p.below.after$ is the ∞-sentinel
  - `// top two lists have only sentinels, remove one`
  - $p.below \leftarrow p.below.below$
  - $p.after.below \leftarrow p.after.below.below$

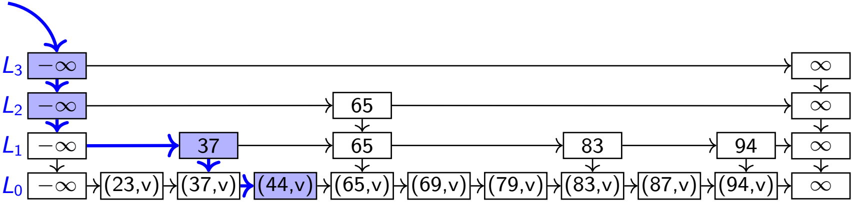
Skip list deletion: Example

Example: *skipList::delete*(65)



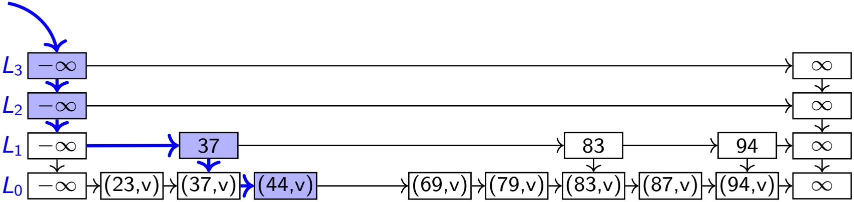
Skip list deletion: Example

Example: *skipList::delete*(65)
get-predecessors(65)



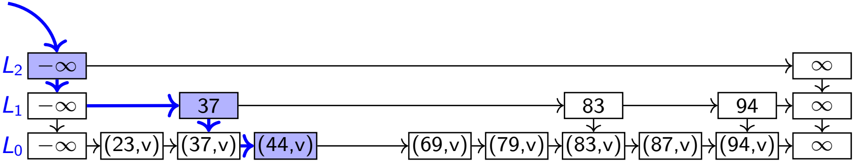
Skip list deletion: Example

Example: *skipList::delete*(65)
get-predecessors(65)



Skip list deletion: Example

Example: *skipList::delete*(65)
get-predecessors(65)
Height decrease



Skip lists: Insertion

skipList::insert(k, v)

- There is no choice as to where to put the tower of k.
- Only choice: how tall should we make the tower of k?
  - Choose *randomly*! Repeatedly toss a coin until you get tails
  - Let i the number of times the coin came up heads
  - We want key k to be in lists $L_0, \ldots, L_i$, so $i =$ *height* of tower of $k =$ how many *extra* lists contain k
  - so

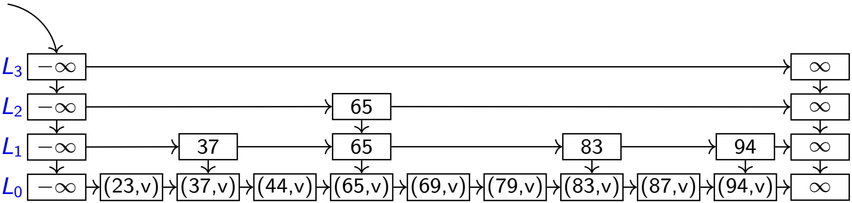
$$Pr(\text{tower of key } k \text{ has height} \geq i) = Pr(\underbrace{H \cdots H}_{i} *) = \left(\tfrac{1}{2}\right)^i$$

- Before we can insert, we must check that these lists exist.
  - Add sentinel-only lists, if needed, until height h satisfies $h > i$.
- Then do the actual insertion.
  - Use *get-predecessors*(k) to get stack P.
  - The top i items of P are the predecessors $p_0, p_1, \cdots, p_i$ of where k should be in each list $L_0, L_1, \cdots, L_i$
  - Insert (k, v) after p_0 in L_0, and k after p_j in L_j for $1 \leq j \leq i$

Skip list insertion: Example

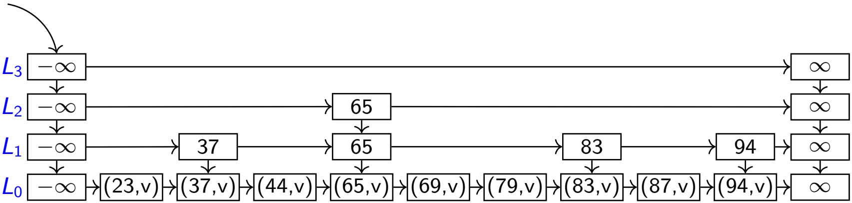
Example: *skipList::insert*$(52, v)$

Coin tosses: H,T $\Rightarrow i = 1$



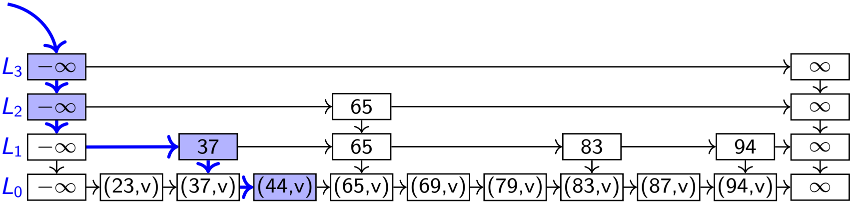
Skip list insertion: Example

Example: *skipList::insert*$(52, v)$
Coin tosses: H,T $\Rightarrow i = 1$
Have $h = 3 > i \Rightarrow$ no need to add lists



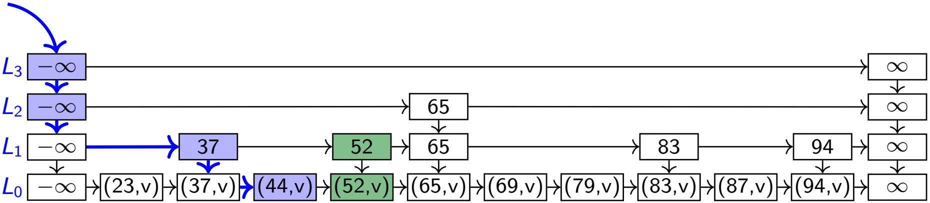
Skip list insertion: Example

Example: *skipList::insert*(52, v)
Coin tosses: H,T $\Rightarrow i = 1$
Have $h = 3 > i \Rightarrow$ no need to add lists
get-predecessors(52)



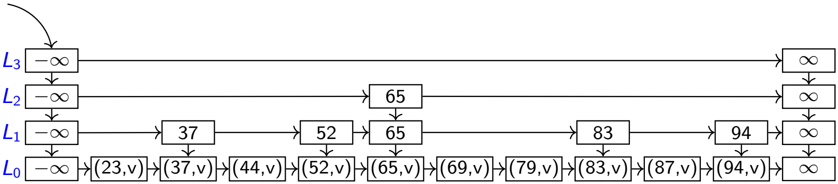
Skip list insertion: Example

Example: *skipList::insert*(52, v)
Coin tosses: H,T $\Rightarrow i = 1$
Have $h = 3 > i \Rightarrow$ no need to add lists
get-predecessors(52)
Insert 52 in lists $L_0, \ldots, L_i$



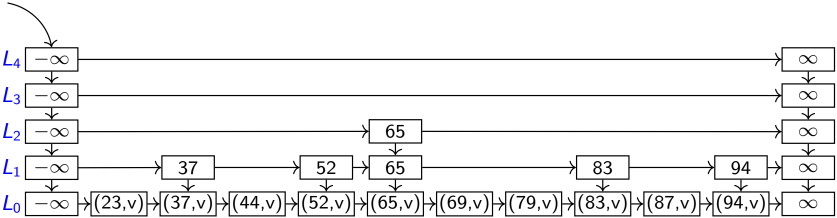
Skip list insert: Example 2

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$



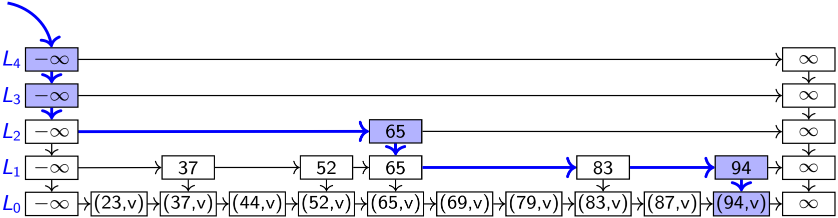
Skip list insert: Example 2

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase



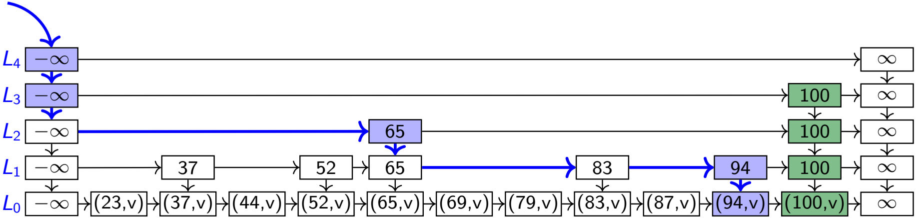
Skip list insert: Example 2

Example: *skipList::insert*$(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase
get-predecessors(100)



Skip list insert: Example 2

Example: *skipList::insert*(100, v)
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase
get-predecessors(100)
Insert 100 in lists $L_0, \ldots, L_i$



Skip list: Insertion

skipList::insert(k, v)

for $(i \leftarrow 0;\ random(2) = 1; i{+}{+})\ \{\}$ `// random tower height`
for $(h \leftarrow 0,\ p \leftarrow root.below; p \neq \texttt{NULL}; p \leftarrow p.below,\ h{+}{+})\ \{\}$
`// compute current height`
while $i \geq h$ `// increase skip-list height?`
  create new sentinel-only list; link it in below topmost list
  $h{+}{+}$
$P \leftarrow$ *get-predecessors*(k)
$p \leftarrow P.pop()$ `// insert` (k, v) `in` L_0
$z_{below} \leftarrow$ new node with (k, v);
$z_{below}.after \leftarrow p.after;\ p.after \leftarrow z_{below}$
while $i > 0$ `// insert` k `in` $L_1, \ldots, L_i$
  $p \leftarrow P.pop()$
  $z \leftarrow$ new node with k
  $z.after \leftarrow p.after;\ p.after \leftarrow z;\ z.below \leftarrow z_{below};\ z_{below} \leftarrow z$
  $i \leftarrow i - 1$

Skip lists: Analysis

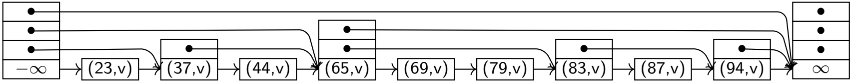
- Expected *space*: $O(\#\text{non-sentinels} + \text{height})$.
  - Expected number of *non-sentinels*? $\mathbf{O(n)}$
  - Expected *height*? $\mathbf{O(\log n)}$

- Run-time of operations is dominated by *get-predecessors*:
  - How often do we *drop down* (execute $p \leftarrow p.below$)? **height**.
  - How often do we *step forward* (execute $p \leftarrow p.after$)?
    Expect $O(1)$ forward-steps per list

So *search*, *insert*, *delete* have $O(\log n)$ expected run-time.

Skip lists: Summary

- $O(n)$ expected space, all operations take $O(\log n)$ expected time.
- Lists make it easy to implement. We can also easily add more operations (e.g. *successor*, *merge*,...)
- Possible improvements on space:
  - Can save links (hence space) by implementing towers as array.



  - Biased coin-flips to determine tower-heights give smaller expected space
  - With both ideas, expected space is $< 2n$

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests
   - Optimal Static Ordering
   - Dynamic Ordering: MTF

Biased search request

Thus far: All keys are assumed to be equal.

- Any key k is equally often the key used when searching.

In reality: Some keys are more frequently accessed than others (**access**: insertion or successful search)

- 80/20 rule: 80% of outcomes result from 20% of causes.
- Rule of **temporal locality**: A recently accessed item is likely to be accessed soon again.
- How can we handle such **biased search requests**?

Intuition: Frequently accessed items should be near the **front** (place where we first search in the data structure).

- Two scenarios: Do we know the access distribution beforehand or not?

Outline

Optimal static ordering

Scenario: We know access distribution, and want the best order of a list.

Example:

key	A	B	C	D	E
number of accesses	2	8	1	10	5

- order $\boxed{D} \to \boxed{B} \to \boxed{E} \to \boxed{A} \to \boxed{C}$ seems better than order $\boxed{A} \to \boxed{B} \to \boxed{C} \to \boxed{D} \to \boxed{E}$
- How do we formalize "better"?
- Define **access frequency** of key $k = \frac{\text{number of accesses of } k}{\text{total number of accesses}}$
- Analyze (for any fixed order of keys) the

$$\textbf{average access cost} = \sum_{i \geq 1} i \cdot (\text{access-frequency of key at position } i)$$

This is proportional to the (weighted) average-case time for *search*.

Optimal static ordering

Example:

key	A	B	C	D	E
number of accesses	2	8	1	10	5
access-frequency	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- Order $\boxed{A} \to \boxed{B} \to \boxed{C} \to \boxed{D} \to \boxed{E}$ has average access cost
$\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31$

- Order $\boxed{D} \to \boxed{B} \to \boxed{E} \to \boxed{A} \to \boxed{C}$ is better!
$\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{54}{26} \approx 2.08$

Claim: Over all possible static orderings, we minimize the average access cost by sorting by non-increasing access-frequency.

Proof:

- Consider any other ordering. How can we improve its access cost?

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests
   - Optimal Static Ordering
   - Dynamic Ordering: MTF

Dynamic ordering: MTF

Scenario: We do *not know the access frequencies* ahead of time.

- **Idea:** modify the order dynamically, i.e., while we are accessing.
- Rule of thumb (**temporal locality**): A recently accessed item is likely to be used soon again.
- **Move-To-Front heuristic** (MTF): Upon a successful search, move the accessed item to the front of the list

A → B → C → D → E

↓ search(D)

D → A → B → C → E

↓ insert(F)

F → D → A → B → C → E

- We can also do MTF on an array, but should then insert and search from the *back* so that we have room to grow.

Dynamic ordering: Other ideas

There are other heuristics we could use:

- **Transpose heuristic**: Upon a successful search, swap the accessed item with the item immediately preceding it

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

$\downarrow$ search(D)

$$A \rightarrow B \rightarrow D \rightarrow C \rightarrow E$$

$\downarrow$ insert(F)

$$F \rightarrow A \rightarrow B \rightarrow D \rightarrow C \rightarrow E$$

  Here the changes are more gradual.

- **Frequency-count heuristic**: Keep counters how often items were accessed, and sort in non-decreasing order.
  Works well in practice, but requires auxiliary space.

Biased search requests: Summary

- We are unlikely to know the access frequencies of items, so optimal static order is mostly of theoretical interest.
- For any dynamic reordering heuristic, some sequence will defeat it (have $\Theta(n)$ access-cost for each item).
- MTF and Frequency-count work well in practice.
- For MTF, can also prove theoretical guarantees.
  - (MTF is an *online* algorithm: Decide based on incomplete information.
  - Compare it to the best *offline* algorithm (has complete information).
  - Here, best offline-algorithm builds optimal static ordering.
  - **Can show:** MTF is "2-competitive": $cost(MTF) \leq 2 \cdot cost(OPT)$.)
- There is very little overhead for MTF and other strategies; they should be applied whenever unordered lists or arrays are used ($\rightarrow$ Hashing, text compression).