CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations

Mark Petrick, Éric Schost
Based on lecture notes by many previous cs240 instructors

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

Spring 2024

version *2024−06−10 13:33*

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests

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**: $\Theta(height)$ search, insert and delete
- **Balanced Binary Search trees** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete

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**: $\Theta(\textit{height})$ 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

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.

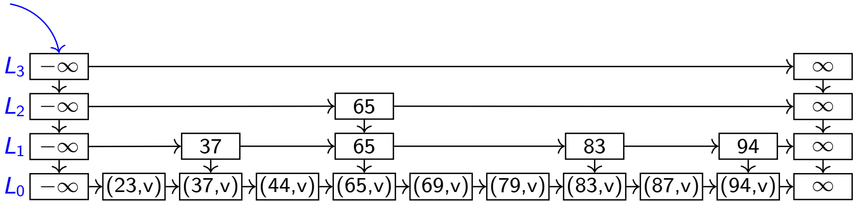
- The bottleneck is *search*:
  - In an ordered array, we can do binary search to achieve $O(\log n)$ run-time.
  - In an ordered list, we cannot 'skip to the middle' and so cannot do binary search.
  - Therefore *search* takes $\Theta(n)$ time in an ordered list—too slow.

Idea: To speed up search in an ordered list, add more links to help us skip forward quicker. Choose randomly when to add such links.

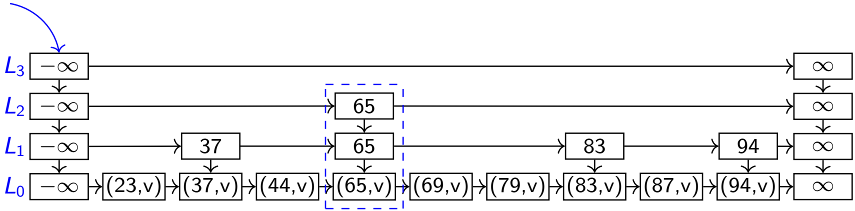
Skip Lists

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



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

Search in Skip Lists

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

get-predecessors (k)

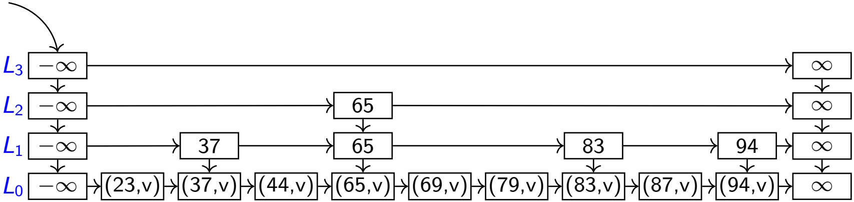
$p \leftarrow$ root
$P \leftarrow$ stack of nodes, initially containing p
while $p.below \neq$ NULL **do**
    $p \leftarrow p.below$
    while $p.after.key < k$ **do** $p \leftarrow p.after$
    $P.push(p)$
return P

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"

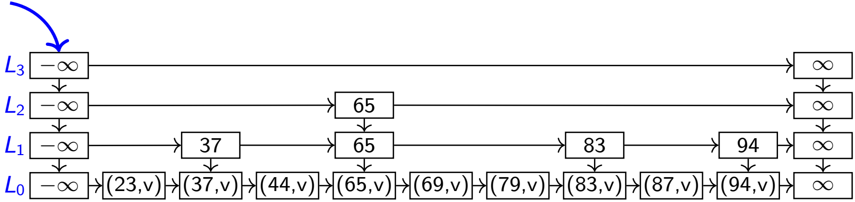
Example: Search in Skip Lists

Example: *search*(87)



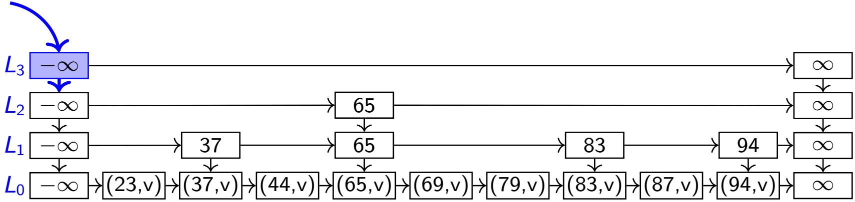
Example: Search in Skip Lists

Example: *search*(87)



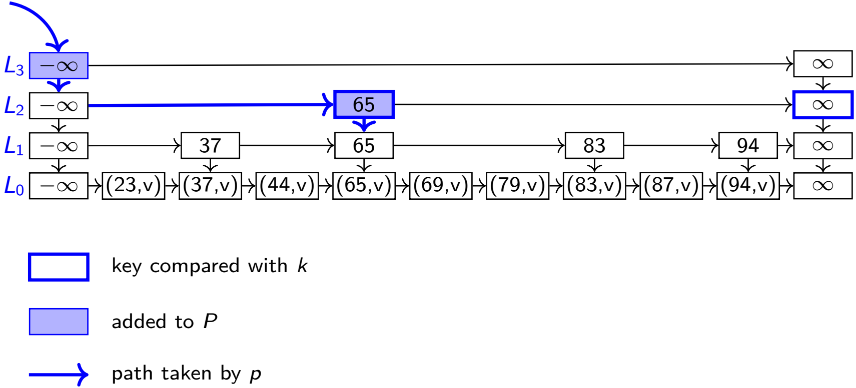
Example: Search in Skip Lists

Example: *search*(87)



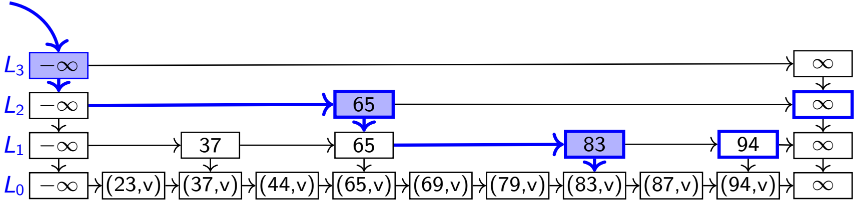
Example: Search in Skip Lists

Example: *search*(87)



Example: Search in Skip Lists

Example: *search*(87)



key compared with k

added to P

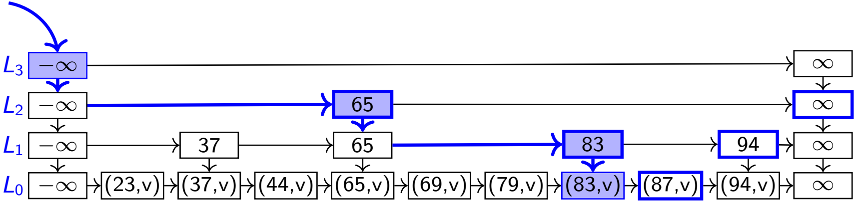
path taken by p

Final stack returned:

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

Example: Search in Skip Lists

Example: *search*(87)




key compared with k


added to P


path taken by p

Final stack returned:

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

Delete in Skip Lists

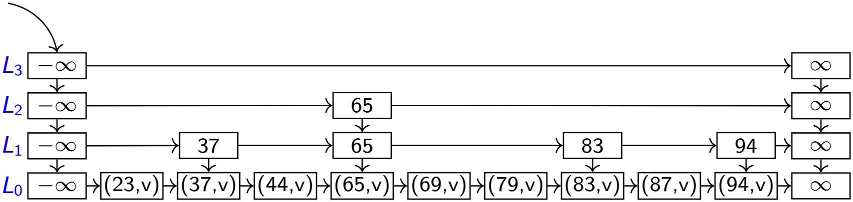
It is easy to remove a key since we can find all predecessors.
Then eliminate lists if there are multiple ones with only sentinels.

```
skipList::delete(k)
P ← get-predecessors(k)
while P is non-empty
        p ← P.pop()            // predecessor of k in some list
        if p.after.key = k
                p.after ← p.after.after
        else break             // no more copies of k

p ← left sentinel of the root-list
while p.below.after is the ∞-sentinel
        // the two top lists are both only sentinels, remove one
        p.below ← p.below.below
        p.after.below ← p.after.below.below
```

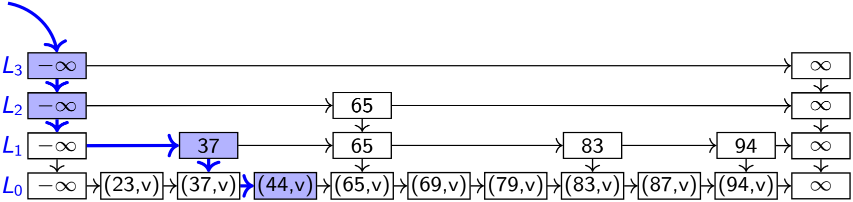
Example: Delete in Skip Lists

Example: *skipList::delete*(65)



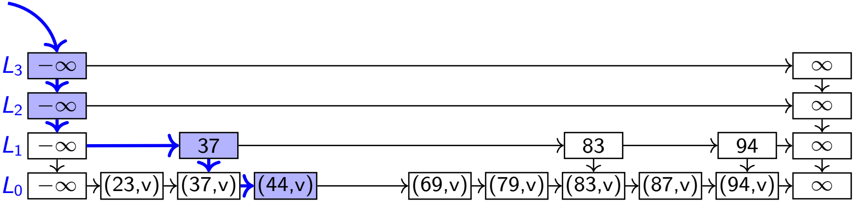
Example: Delete in Skip Lists

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



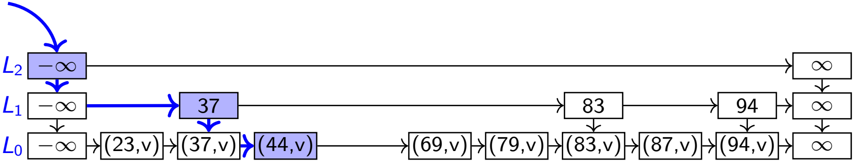
Example: Delete in Skip Lists

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



Example: Delete in Skip Lists

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



Insert in Skip Lists

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 $i \rightarrow$ *height* of tower of k

$$P(\text{tower of key } k \text{ has height} \geq i) = \left(\tfrac{1}{2}\right)^i$$

Insert in Skip Lists

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 $i \rightarrow$ *height* of tower of k

$$P(\text{tower of key } k \text{ has height} \geq i) = \left(\tfrac{1}{2}\right)^i$$

- Before we can insert, we must check that these lists exist.
  - Add sentinel-only lists, if needed, until height h satisfies $h > i$.

Insert in Skip Lists

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 $i \rightarrow$ *height* of tower of k

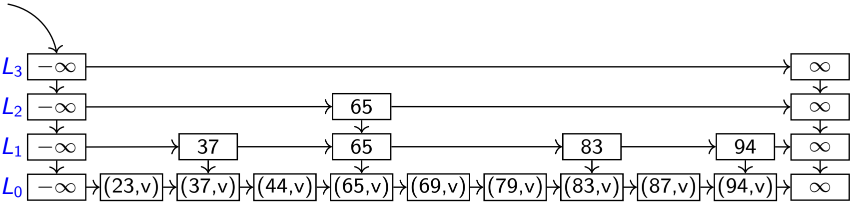
$$P(\text{tower of key } k \text{ has height} \geq i) = \left(\tfrac{1}{2}\right)^i$$

- 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 $1 \leq j \leq i$

Example: Insert in Skip Lists

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

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

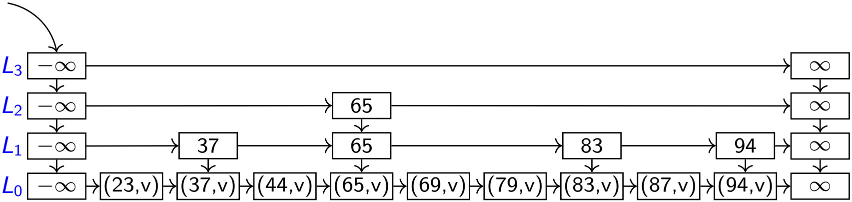


Example: Insert in Skip Lists

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

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

Have $h = 3 > i \Rightarrow$ no need to add lists



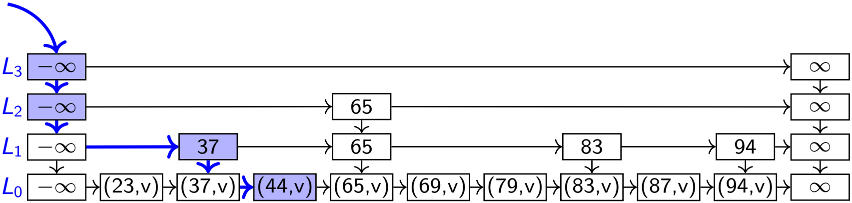
Example: Insert in Skip Lists

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)



Example: Insert in Skip Lists

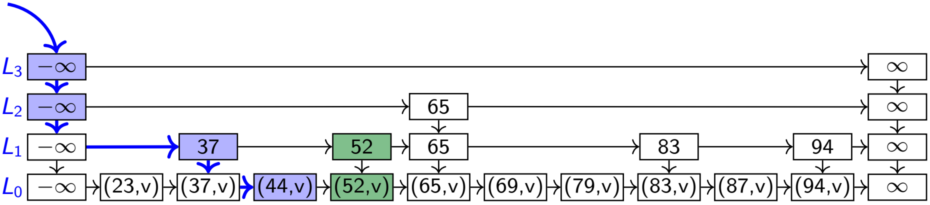
Example: *skipList::insert*(52, v)

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

Have $h = 3 > i \Rightarrow$ no need to add lists

get-predecessors(52)

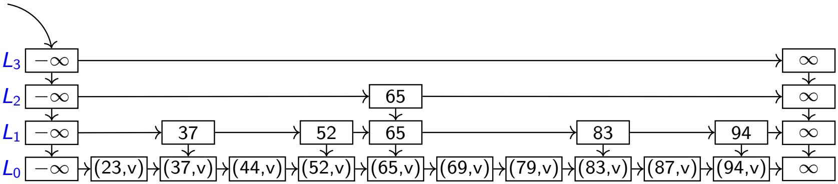
Insert 52 in lists $L_0, \ldots, L_i$



Example 2: Insert in Skip Lists

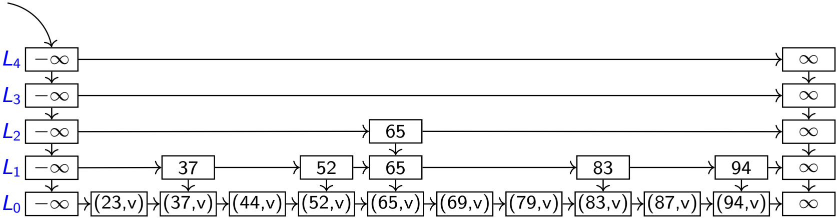
Example: *skipList::insert*(100, v)

Coin tosses: H,H,H,T $\Rightarrow i = 3$



Example 2: Insert in Skip Lists

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



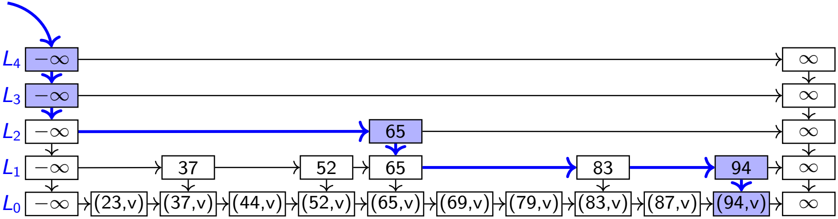
Example 2: Insert in Skip Lists

Example: *skipList::insert*(100, v)

Coin tosses: H,H,H,T $\Rightarrow i = 3$

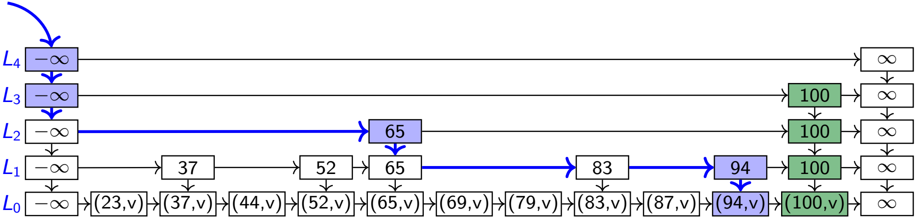
Height increase

get-predecessors(100)



Example 2: Insert in Skip Lists

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



Insert in Skip Lists

skipList::insert(k, v)

```
for (i ← 0; random(2) = 1; i++) {}        // random tower height
for (h ← 0, p ← root.below; p ≠ NULL; p ← p.below, h++) {}
while i ≥ h                               // increase skip-list height?
    create new sentinel-only list; link it in below topmost list
    h++
P ← get-predecessors(k)
p ← P.pop()                               // insert (k, v) in L_0
z_below ← new node with (k, v);
z_below.after ← p.after; p.after ← z_below
while i > 0                               // insert k in L_1, ..., L_i
    p ← P.pop()
    z ← new node with k
    z.after ← p.after; p.after ← z; z.below ← z_below; z_below ← z
    i ← i − 1
```

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
  - Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
  - Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.
  - $E[|L_i|] = \ldots$
  - $E[\#\text{non-sentinels}] = \sum_{i=0}^{h} E[|L_i|] = \ldots$

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
  - Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.
  - $E[|L_i|] = \ldots$
  - $E[\#\text{non-sentinels}] = \sum_{i=0}^{h} E[|L_i|] = \ldots$
- Expected **height**: $O(\log n)$. [Similar (longer) proof omitted.]

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
  - Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.
  - $E[|L_i|] = \ldots$
  - $E[\#\text{non-sentinels}] = \sum_{i=0}^{h} E[|L_i|] = \ldots$

- Expected **height**: $O(\log n)$. [Similar (longer) proof omitted.]

- *skipList::get-predecessors*: $O(\log n)$ expected time
  - How often do we *drop down* (execute $p \leftarrow p.below$)? *height*.
  - How often do we *step forward* (execute $p \leftarrow p.after$)?
    Can show: expect to step forward at most once in each list

Analysis of Skip Lists

- Expected **space** usage: $O(n)$
  - Set X_k = tower height of key k. Recall $\Pr(X_k \geq i) = \left(\frac{1}{2}\right)^i$.
  - Define $|L_i|$ = #non-sentinels in L_i. Observe $|L_i| = \sum_k \chi_{(X_k \geq i)}$.
  - $E[|L_i|] = \dots$
  - $E[\#\text{non-sentinels}] = \sum_{i=0}^{h} E[|L_i|] = \dots$

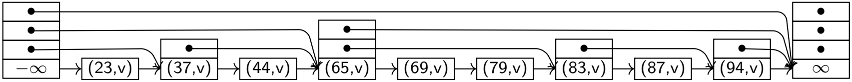
- Expected **height**: $O(\log n)$. [Similar (longer) proof omitted.]

- *skipList::get-predecessors*: $O(\log n)$ expected time
  - How often do we *drop down* (execute $p \leftarrow p.below$)? *height*.
  - How often do we *step forward* (execute $p \leftarrow p.after$)?
    Can show: expect to step forward at most once in each list

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

Summary of Skip Lists

- $O(n)$ 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*,...)
- As described they are no better than randomized binary search trees.
- But there are numerous improvements on the 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 $< 2n$ (less than for a BST).

Outline

5. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Biased Search Requests

Improving unsorted lists/arrays

Recall *unsorted array* realization:

0	1	2	3	4
90	30	60	20	50

- *search*: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Very simple and popular. Can we do something to make search more effective in practice?

Improving unsorted lists/arrays

Recall *unsorted array* realization:

0	1	2	3	4
90	30	60	20	50

- *search*: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Very simple and popular. Can we do something to make search more effective in practice?
- No: if items are accessed equally likely.
  We can show that the average-case cost for *search* is then $\Theta(n)$.
- Yes: if the search requests are **biased**:
  some items are accessed much more frequently than others.
  - 80/20 rule: 80% of outcomes result from 20% of causes.
  - **access**: insertion or successful search
  - Intuition: Frequently accessed items should be in the front.
  - Two scenarios: Do we know the access distribution beforehand or not?

Optimal Static Ordering

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

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5

Recall: $T^{avg}(n) = \sum_{I \in \mathcal{I}_n} T(I) \cdot (\text{relative frequency of } I)$

$= \text{expected run-time on randomly chosen input}$

$= \sum_{I \in \mathcal{I}_n} T(I) \cdot \Pr(\text{randomly chosen instance is } I)$

Optimal Static Ordering

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

Example:

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

Recall:
$$\begin{aligned} T^{avg}(n) &= \textstyle\sum_{I \in \mathcal{I}_n} T(I) \cdot (\text{relative frequency of } I) \\ &= \text{expected run-time on randomly chosen input} \\ &= \textstyle\sum_{I \in \mathcal{I}_n} T(I) \cdot \Pr(\text{randomly chosen instance is } I) \end{aligned}$$

- Count cost i if search-key ($=$ instance I) is at ith position ($i \geq 1$).
- $T^{avg}(n)$=**expected access cost** $= \sum_{i \geq 1} i \cdot \underbrace{\Pr\left(\text{search for key at position } i\right)}_{\textbf{access-probability} \text{ of that key}}$
- Example: Order A → B → C → D → E has expected access cost
  $\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31$

Optimal Static Ordering

- Order A → B → C → D → E has expected 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 D → B → E → A → C is better!
  $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54$

Claim: Over all possible static orderings, the one that sorts items by non-increasing access-probability minimizes the expected access cost.

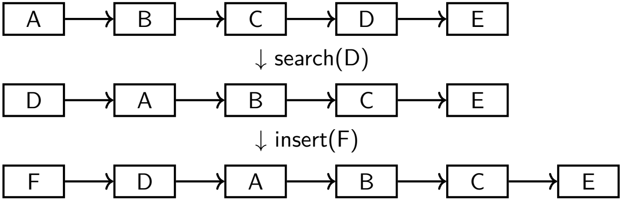
Proof:

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

Dynamic Ordering: MTF

Scenario: We do *not know the access probabilities* 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

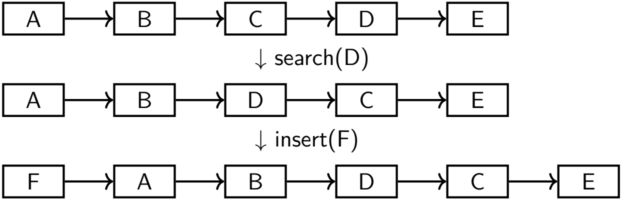


- We can also do MTF on an array, but should then insert and search from the *back* so that we have room to grow.

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

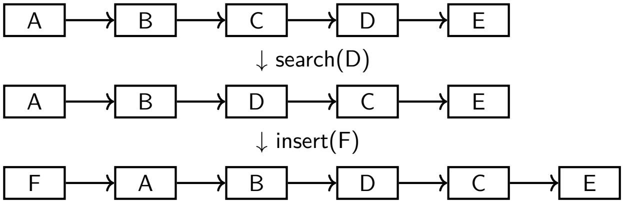


Here the changes are more gradual.

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



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.

Summary of biased search requests

- We are unlikely to know the access-probabilities 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.

Summary of biased search requests

- We are unlikely to know the access-probabilities 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": $cost(MTF) \leq 2 \cdot cost(OPT)$.

)

Summary of biased search requests

- We are unlikely to know the access-probabilities 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).