

# 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 2025

# 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:**  $\Theta(\text{height})$  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:**  $\Theta(\text{height})$  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?

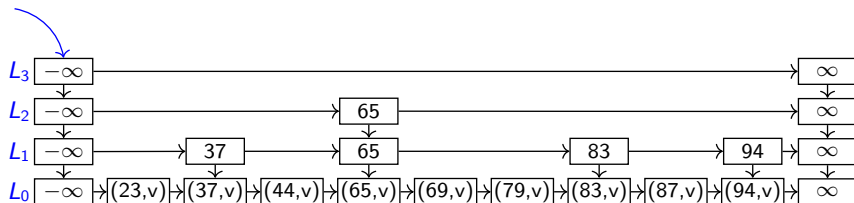
- *insert* and *delete* take  $\Theta(1)$  time in an ordered lists, once we know the place where to do them.
- The bottleneck is *search*:
  - ▶ In an ordered array, we can do binary search to achieve  $O(\log n)$  run-time.
  - ▶ In an ordered list, we cannot 'skip to the middle' and so cannot do binary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

**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

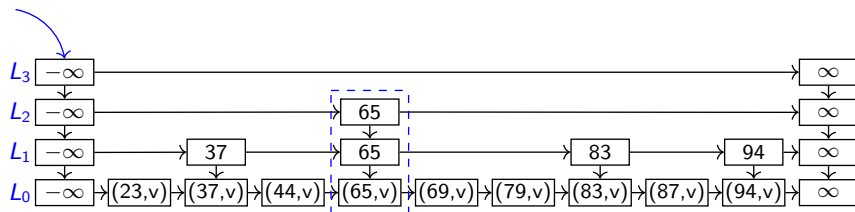
A hierarchy of ordered linked lists (*levels*)  $L_0, L_1, \dots, L_h$ :

- 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.,  
 $L_0 \supseteq L_1 \supseteq \dots \supseteq L_h$
- 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* ( $k$ )

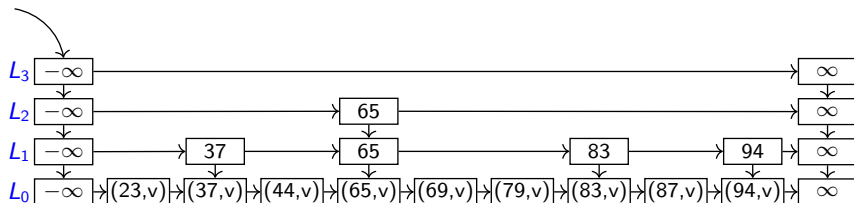
1.  $p \leftarrow \text{root}$
2.  $P \leftarrow$  stack of nodes, initially containing  $p$
3. **while**  $p.\text{below} \neq \text{NULL}$  **do**
4.      $p \leftarrow p.\text{below}$
5.     **while**  $p.\text{after.key} < k$  **do**  $p \leftarrow p.\text{after}$
6.      $P.\text{push}(p)$
7. **return**  $P$

*skipList::search* ( $k$ )

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

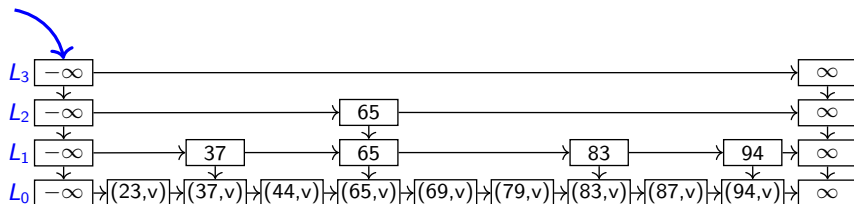
# Skip list search: Example

**Example:** *search*(87)



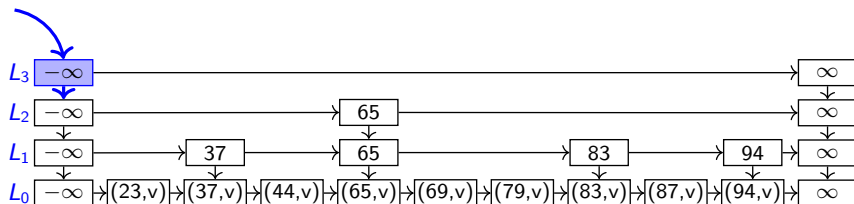
# Skip list search: Example

**Example:** *search*(87)



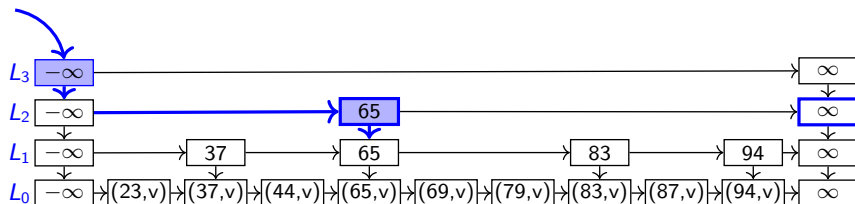
# Skip list search: Example

**Example:** *search*(87)



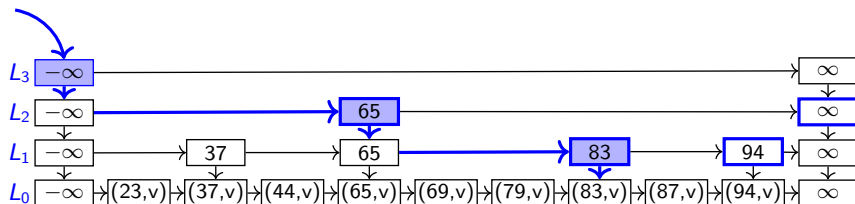
# Skip list search: Example

**Example:** *search*(87)



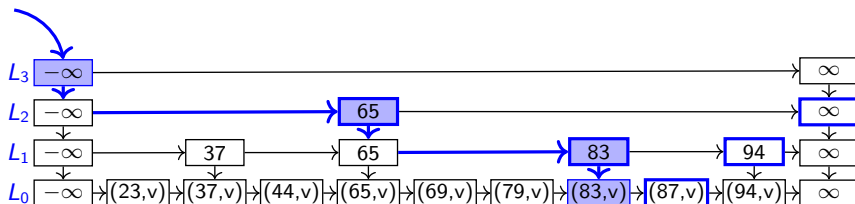
# Skip list search: Example

**Example:** *search*(87)



# Skip list search: Example

**Example:** *search*(87)



Final stack returned:

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



## Skip list: Deletion

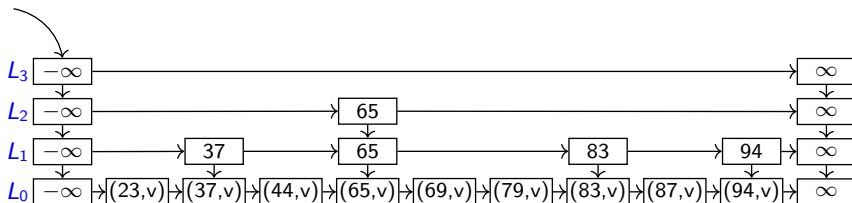
It is easy to remove a key since we can find all predecessors.  
Then eliminate lists if there are multiple ones with only sentinels.

```
skipList::delete(k)
1.  $P \leftarrow \text{get-predecessors}(k)$ 
2. while  $P$  is non-empty
3.      $p \leftarrow P.\text{pop}()$  // predecessor of  $k$  in some list
4.     if  $p.\text{after.key} = k$ 
5.          $p.\text{after} \leftarrow p.\text{after}.\text{after}$ 
6.     else break // no more copies of  $k$ 

7.  $p \leftarrow$  left sentinel of the root-list
8. while  $p.\text{below.after}$  is the  $\infty$ -sentinel
    // top two lists have only sentinels, remove one
9.      $p.\text{below} \leftarrow p.\text{below}.\text{below}$ 
10.     $p.\text{after.below} \leftarrow p.\text{after.below.below}$ 
```

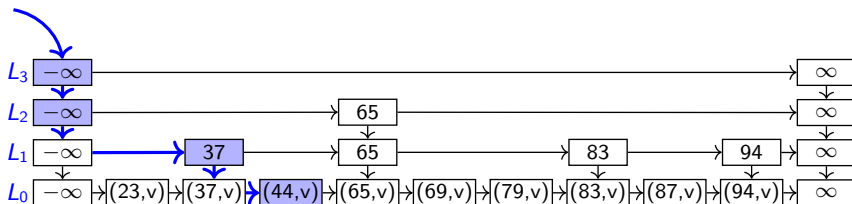
# Skip list deletion: Example

Example: *skipList::delete*(65)



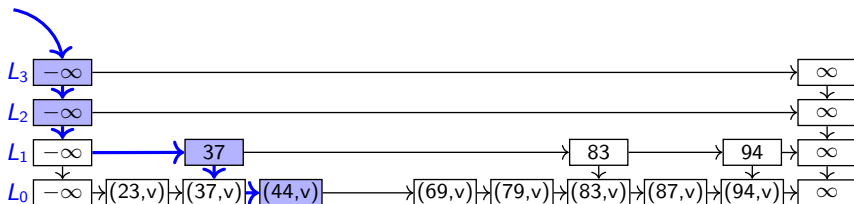
# Skip list deletion: Example

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



# Skip list deletion: Example

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

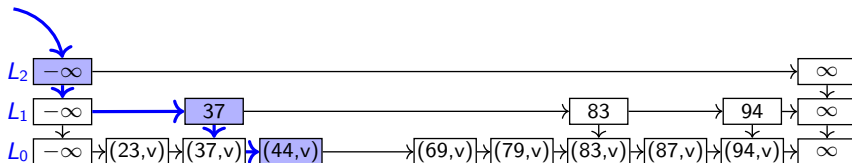


# Skip list deletion: Example

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  $L_0, \dots, L_i$ , so  $i \rightarrow$  *height* of tower of  $k$

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

# 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  $L_0, \dots, L_i$ , so  $i \rightarrow$  *height* of tower of  $k$

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

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

# 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  $L_0, \dots, L_i$ , so  $i \rightarrow$  *height* of tower of  $k$

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

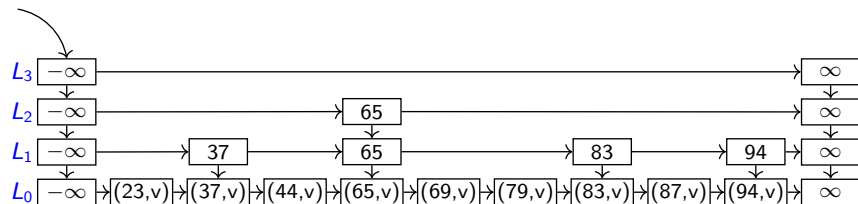
- Before we can insert, we must check that these lists exist.
  - ▶ Add sentinel-only lists, if needed, until height  $h$  satisfies  $h > i$ .
- Then do the actual insertion.
  - ▶ Use *get-predecessors*( $k$ ) to get stack  $P$ .
  - ▶ The top  $i$  items of  $P$  are the predecessors  $p_0, p_1, \dots,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

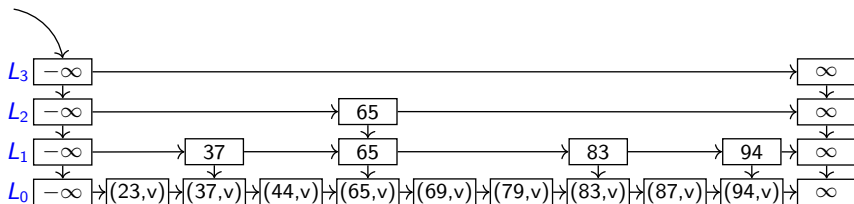


# Skip list insertion: Example

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

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

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



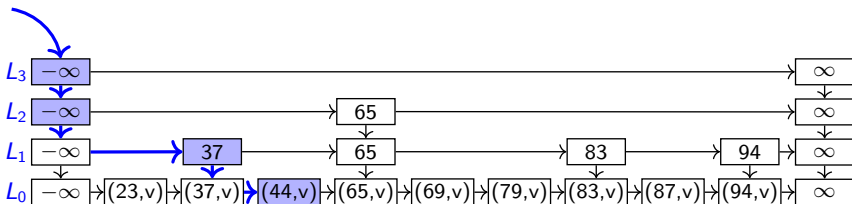
# Skip list insertion: Example

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

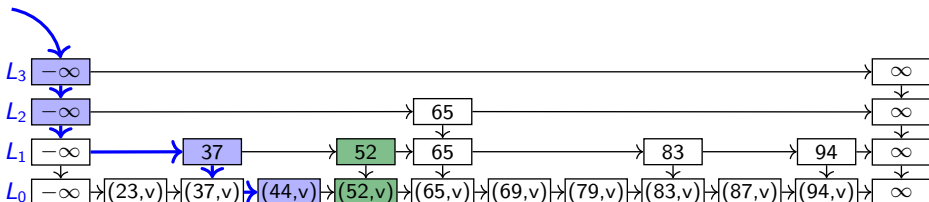
Example: *skipList::insert*(52,  $v$ )

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

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

*get-predecessors*(52)

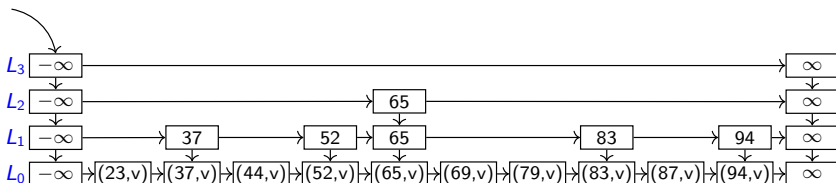
Insert 52 in lists  $L_0, \dots, L_i$



## Skip list insert: Example 2

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

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

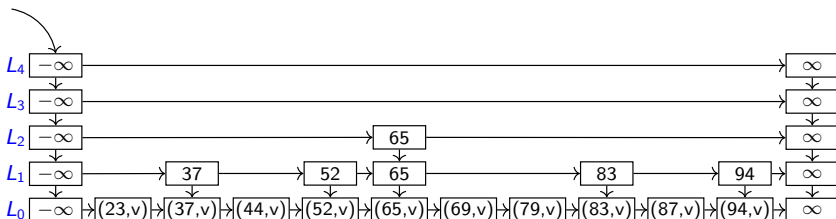


## Skip list insert: Example 2

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

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

*Height increase*



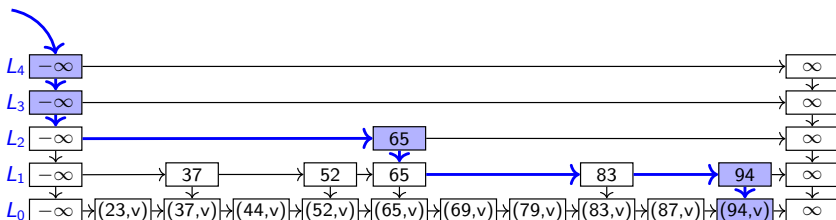
## Skip list insert: Example 2

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

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

*Height increase*

*get-predecessors*(100)



## Skip list insert: Example 2

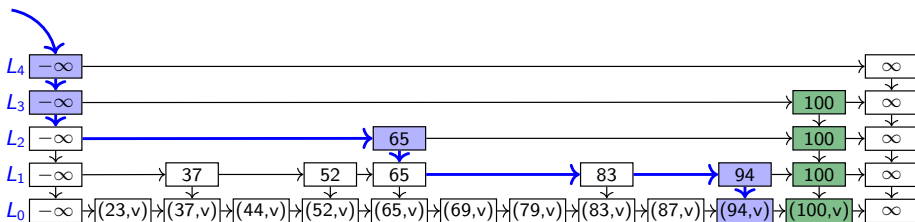
Example: *skipList::insert*(100,  $v$ )

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

*Height increase*

*get-predecessors*(100)

Insert 100 in lists  $L_0, \dots, L_i$





# Skip list: Insertion

*skipList::insert*( $k, v$ )

1. **for** ( $i \leftarrow 0$ ; *random*(2) = 1;  $i++$ ) {} // random tower height
2. **for** ( $h \leftarrow 0$ ,  $p \leftarrow \text{root.below}$ ;  $p \neq \text{NULL}$ ;  $p \leftarrow p.\text{below}$ ,  $h++$ ) {}
3. **while**  $i \geq h$  // increase skip-list height?
4.     create new sentinel-only list; link it in below topmost list
5.      $h++$
6.    $P \leftarrow \text{get-predecessors}(k)$
7.    $p \leftarrow P.\text{pop}()$  // insert ( $k, v$ ) in  $L_0$
8.    $z_{\text{below}} \leftarrow$  new node with ( $k, v$ );
9.    $z_{\text{below}}.\text{after} \leftarrow p.\text{after}$ ;  $p.\text{after} \leftarrow z_{\text{below}}$
10. **while**  $i > 0$  // insert  $k$  in  $L_1, \dots, L_i$
11.    $p \leftarrow P.\text{pop}()$
12.    $z \leftarrow$  new node with  $k$
13.    $z.\text{after} \leftarrow p.\text{after}$ ;  $p.\text{after} \leftarrow z$ ;  $z.\text{below} \leftarrow z_{\text{below}}$ ;  $z_{\text{below}} \leftarrow z$
14.    $i \leftarrow i - 1$

## Skip lists: Analysis

- Expected *space*:  $O(\# \text{non-sentinels} + \text{height})$ .

# Skip lists: Analysis

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

# Skip lists: Analysis

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

# Skip lists: Analysis

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

# Skip lists: Analysis

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

So expected space is  $O(n)$ .

# Skip lists: Analysis

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

So expected space is  $O(n)$ .

- Run-time of operations is dominated by *get-predecessors*:
  - ▶ How often do we *drop down* (execute  $p \leftarrow p.\text{below}$ )? **height**.
  - ▶ How often do we *step forward* (execute  $p \leftarrow p.\text{after}$ )?

# Skip lists: Analysis

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

So expected space is  $O(n)$ .

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



# Skip lists: Analysis

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

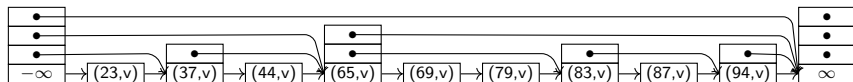
So expected space is  $O(n)$ .

- Run-time of operations is dominated by *get-predecessors*:
  - ▶ How often do we *drop down* (execute  $p \leftarrow p.\text{below}$ )? **height**.
  - ▶ How often do we *step forward* (execute  $p \leftarrow p.\text{after}$ )?  
**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*,...)
- 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**
- 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.

# 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**?

# 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

## 5 Dictionaries with Lists revisited

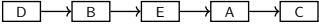
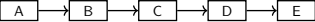
- Dictionary ADT: Implementations thus far
- Skip Lists
- Biased Search Requests
- **Optimal Static Ordering**
- Dynamic Ordering: MTF

# 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  seems better than order 
- How do we formalize “better”?

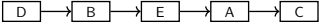
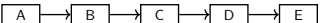


# Optimal static ordering

**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  seems better than order 
- How do we formalize “better”?
- Define **access probability** of key  $k = \frac{\text{number of accesses of } k}{\text{total number of accesses}}$
- Analyze (for any fixed order of keys) the

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

(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-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

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

# Optimal static ordering

## Example:

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

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

**Claim:** Over all possible static orderings, we minimize the expected access cost by sorting by non-increasing access-probability.

## Proof:

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

# Outline

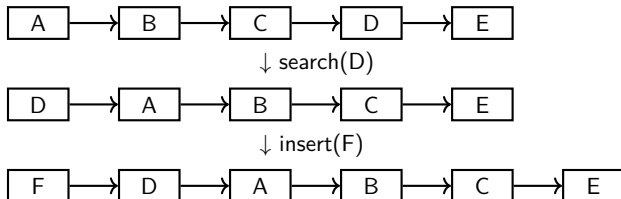
## 5 Dictionaries with Lists revisited

- 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 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

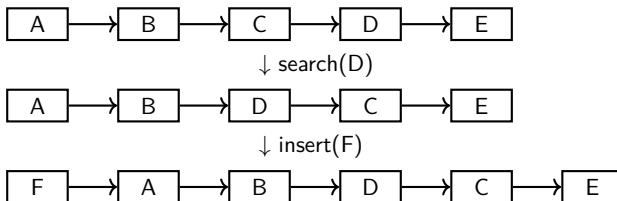


- We can also do MTF on an array, but should then insert and search from the *back* so that we have room to grow.

## 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

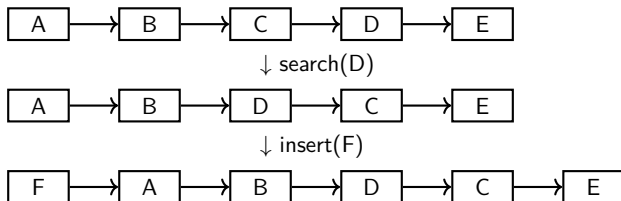


Here the changes are more gradual.

## 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.

## Biased search requests: Summary

- 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.



## Biased search requests: Summary

- 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.
- $\left( \begin{array}{l} \blacktriangleright \text{MTF is an } \textcolor{red}{\textit{online}} \text{ algorithm: Decide based on incomplete information.} \\ \blacktriangleright \text{Compare it to the best } \textcolor{red}{\textit{offline}} \text{ algorithm (has complete information).} \\ \blacktriangleright \text{Here, best offline-algorithm builds optimal static ordering.} \\ \blacktriangleright \textcolor{violet}{\text{Can show:}} \text{ MTF is "2-competitive": } \textit{cost}(\textit{MTF}) \leq 2 \cdot \textit{cost}(\textit{OPT}). \end{array} \right)$

## Biased search requests: Summary

- 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”:  $\text{cost}(\text{MTF}) \leq 2 \cdot \text{cost}(\text{OPT})$ .)
- 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).