Module 6: Dictionary Tricks

CS 240 - Data Structures and Data Management

Therese Biedl   Daniela Maftuleac

Based on lecture notes by many previous cs240 instructors

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

Winter 2015

Dictionary ADT: Review

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

Realizations

- **Unordered array or linked list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Balanced search trees** (AVL trees, 2-3 trees):
  $\Theta(\log n)$ search, insert, and delete
- **Hash tables** (on average, under UHA):
  $\Theta(1)$ search, insert, and delete

Interpolation Search

Ordered array

- *insert, delete*: $\Theta(n)$
- *search*: $\Theta(\log n)$

binary search$(A[\ell, r], k)$: Check index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r - \ell) \rfloor$

Question: What if the keys are numbers?

Interpolation Search

Ordered array

- *insert, delete*: $\Theta(n)$
- *search*: $\Theta(\log n)$

binary search$(A[\ell, r], k)$: Check index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r-\ell) \rfloor$

Question: What if the keys are numbers?

Idea: Use the value of the key to *guess* its location

Interpolation Search$(A[\ell, r], k)$: Check index $\ell + \lfloor \frac{k - A[\ell]}{A[r]-A[\ell]}(r-\ell) \rfloor$

Works well if keys are uniformly distributed: $O(\log \log n)$ on average.
Bad worst case performance: $O(n)$

Gallop Search

Problem in Binary-Search: Sometimes we cannot see the end of the array (data streams, a huge file, etc.)

Gallop-Search(A, k)
A: An ordered array, k: a key
1. $i \leftarrow 0$
2. **while** $i < \text{size}(A)$ **and** $k > A[i]$ **do**
3. $\quad i \leftarrow 2i + 1$
4. **return** Binary-Search$(A\big[\lceil i/2 \rceil, \min(i, \text{size}(A) - 1)\big], k)$

$O(\log m)$ comparisons (m: location of k in A)

Self-Organizing Search

- Unordered linked list
  search: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Linear search to find an item in the list
- Is there a more useful ordering?

Self-Organizing Search

- Unordered linked list
  search: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Linear search to find an item in the list
- Is there a more useful ordering?

- No: if items are accessed equally likely
- Yes: otherwise (we have a probability distribution for items)

- Optimal static ordering: sorting items by their probabilities of access in non-increasing order minimizes the expected cost of Search.
- Proof Idea: For any other ordering, exchanging two items that are out-of-order according to their access probabilities makes the total cost decrease.

Dynamic Ordering

- What if we do not know the access probabilities ahead of time?
- Move-To-Front(MTF): Upon a successful search, move the accessed item to the front of the list
- Transpose: Upon a successful search, swap the accessed item with the item immediately preceding it

Dynamic Ordering

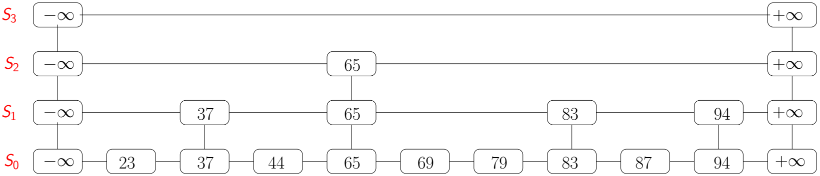
- What if we do not know the access probabilities ahead of time?
- Move-To-Front(MTF): Upon a successful search, move the accessed item to the front of the list
- Transpose: Upon a successful search, swap the accessed item with the item immediately preceding it

Performance of dynamic ordering:

- Both can be implemented in arrays or linked lists.
- Transpose can perform very badly on certain input.
- MTF Works well in practice.
- Theoretically MTF is "competitive":
  No more than twice as bad as the optimal "offline" ordering.

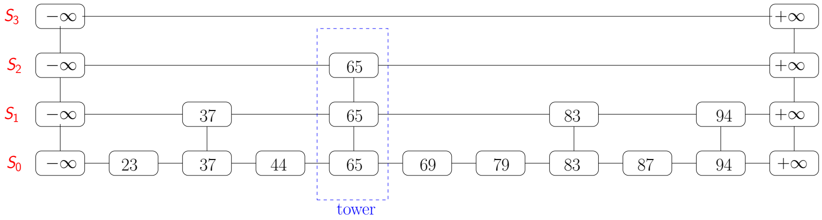
Skip Lists

- Randomized data structure for dictionary ADT
- A hierarchy of ordered linked lists
- A skip list for a set S of items is a series of lists $S_0, S_1, \cdots, S_h$ such that:
  - Each list S_i contains the special keys $-\infty$ and $+\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$
  - List S_h contains only the two special keys



Skip Lists

- A skip list for a set S of items is a series of lists $S_0, S_1, \cdots, S_h$
- A two-dimensional collection of positions: levels and towers
- Traversing the skip list: after(p), below(p)



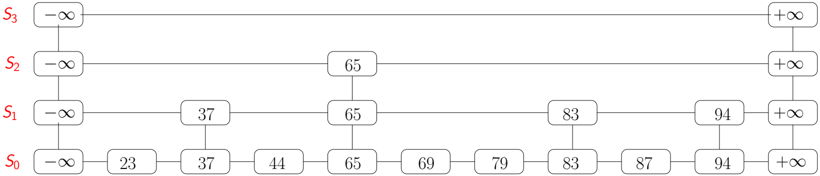
Search in Skip Lists

```
skip-search(L, k)
L: A skip list, k: a key
    p ← topmost left position of L
    S ← stack of positions, initially containing p
    while below(p) ≠ null do
        p ← below(p)
        while key(after(p)) < k do
            p ← after(p)
        push p onto S
    return S
```

- S contains positions of the largest key **less than** k at each level.
- $after(top(S))$ will have key k, iff k is in L.
- drop down: $p \leftarrow below(p)$
- scan forward: $p \leftarrow after(p)$

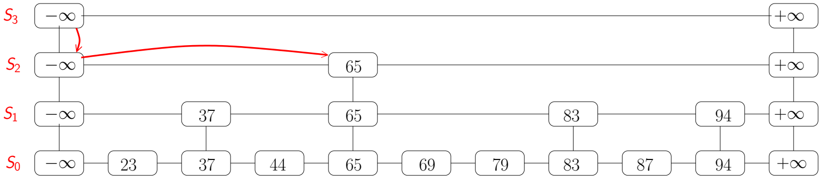
Search in Skip Lists

Example: Skip-Search(S, 87)



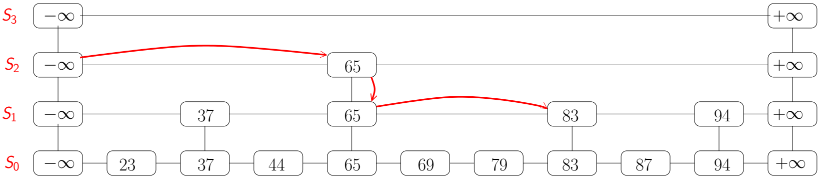
Search in Skip Lists

Example: Skip-Search$(S, 87)$



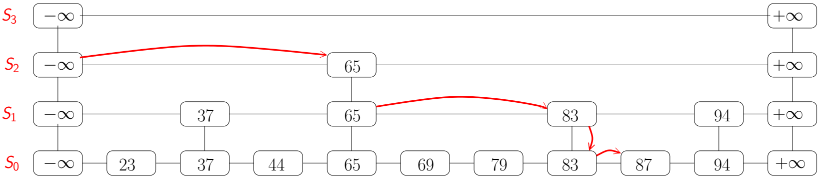
Search in Skip Lists

Example: Skip-Search(S, 87)



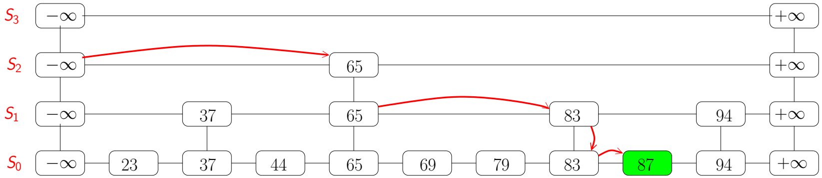
Search in Skip Lists

Example: Skip-Search$(S, 87)$



Search in Skip Lists

Example: Skip-Search(S, 87)

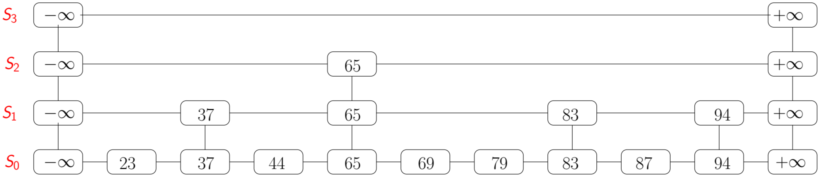


Insert in Skip Lists

- *Skip-Insert*(S, k, v)
  - Randomly compute the height of new item: repeatedly toss a coin until you get tails, let i the number of times the coin came up heads
  - Search for k in the skip list and find the positions $p_0, p_1, \cdots, p_i$ of the items with largest key less than k in each list $S_0, S_1, \cdots, S_i$ (by performing *Skip-Search*(S, k))
  - Insert item (k, v) into list S_j after position p_j for $0 \leq j \leq i$ (a tower of height i)

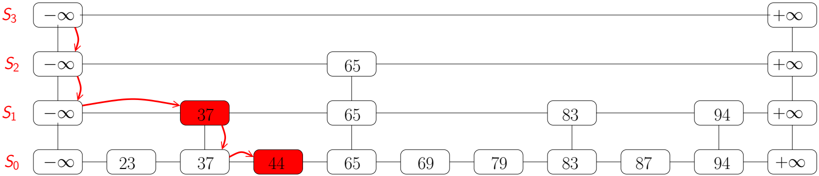
Insert in Skip Lists

Example: Skip-Insert($S, 52, v$)
Coin tosses: H,T $\Rightarrow i = 1$



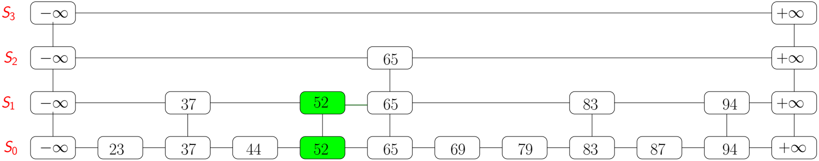
Insert in Skip Lists

Example: Skip-Insert$(S, 52, v)$
Coin tosses: H,T $\Rightarrow i = 1$
Skip-Search$(S, 52)$



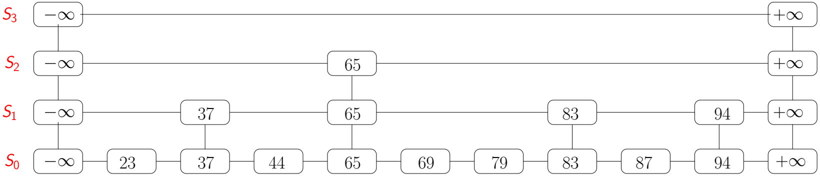
Insert in Skip Lists

Example: Skip-Insert$(S, 52, v)$
Coin tosses: H,T $\Rightarrow i = 1$



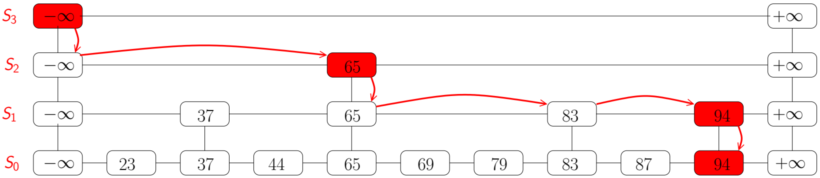
Insert in Skip Lists

Example: Skip-Insert($S, 100, v$)
Coin tosses: H,H,H,T $\Rightarrow i = 3$



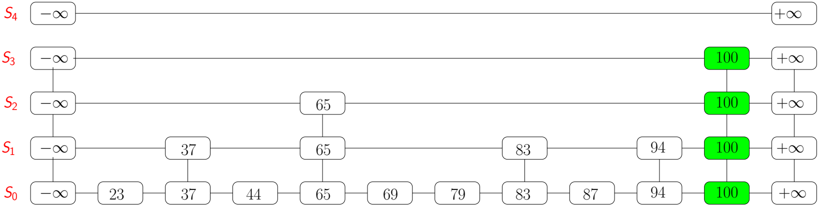
Insert in Skip Lists

Example: Skip-Insert$(S, 100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Skip-Search$(S, 100)$



Insert in Skip Lists

Example: Skip-Insert$(S, 100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Height increase

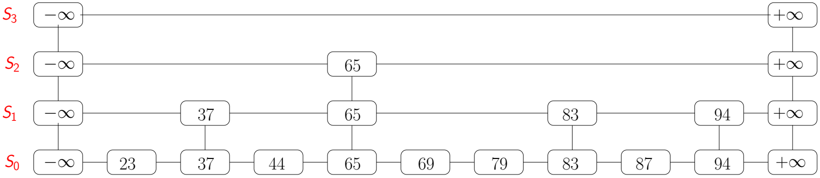


Delete in Skip Lists

- Skip-Delete(S, k)
  - Search for k in the skip list and find all the positions $p_0, p_1, \ldots, p_i$ of the items with the largest key smaller than k, where p_j is in list S_j. (this is the same as Skip-Search)
  - For each i, if $key(after(p_i)) == k$, then remove $after(p_i)$ from list S_i
  - Remove all but one of the lists S_i that contain only the two special keys

Delete in Skip Lists

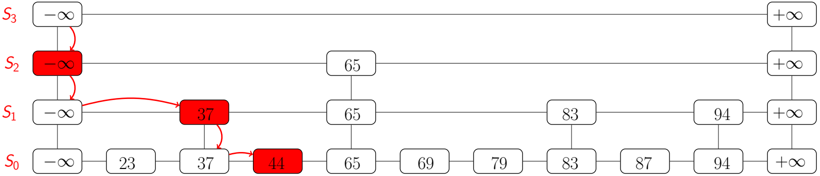
Example: Skip-Delete(S, 65)



Delete in Skip Lists

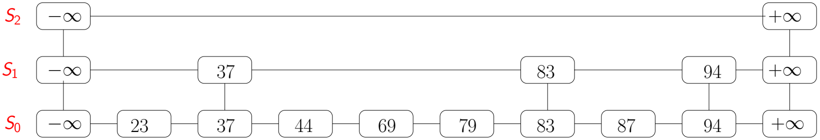
Example: Skip-Delete(S, 65)

Skip-Search(S, 65)



Delete in Skip Lists

Example: Skip-Delete(S, 65)



Summary of Skip Lists

- Expected space usage: $O(n)$
- Expected height: $O(\log n)$
  A skip list with n items has height at most $3 \log n$ with probability at least $1 - 1/n^2$
- *Skip-Search*: $O(\log n)$ expected time
- *Skip-Insert*: $O(\log n)$ expected time
- *Skip-Delete*: $O(\log n)$ expected time
- Skip lists are fast and simple to implement in practice