CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations

A. Storjohann
Based on lecture notes by many previous cs240 instructors

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

Fall 2018

References: Sedgewick 13.5, 12.4

version *2018–10–09 09:50*

Outline

1. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Re-ordering Items

Outline

1. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Re-ordering Items

Dictionary ADT: Implementations thus far

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

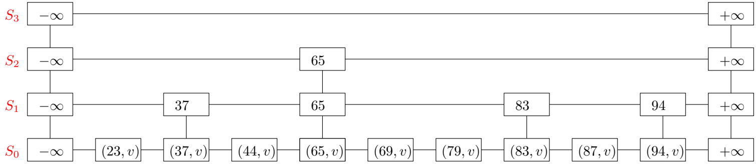
Improvements/Simplifications?

- **Can show:** The average-case height of binary search trees (over all possible insertion sequences) is $O(\log n)$.
- How can we shift the average-case to expected height via randomization?

Outline

Skip Lists

- A hierarchy S of ordered linked lists (*levels*) $S_0, S_1, \cdots, S_h$:
  - Each list S_i contains the special keys $-\infty$ and $+\infty$ (sentinels)
  - List S_0 contains the KVPs of S in non-decreasing order. (The other lists store only keys, or links to nodes in S_0.)
  - 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 sentinels



- The skip list consists of a reference to the topmost left node.
- Each node p has references to after(p), below(p)
- Each KVP belongs to a *tower* of nodes

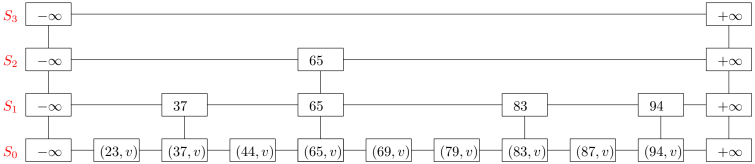
Search in Skip Lists

```
skip-search(L, k)
    p ← topmost left node of L
    P ← stack of nodes, initially containing p
    while below(p) ≠ null do
        p ← below(p)
        while key(after(p)) < k do
            p ← after(p)
        push p onto P
    return P
```

- P collects **predecessors** of k at level $S_0, S_1, \ldots$
  (These will be needed for insert/delete.)
- k is in L if and only if *after*(*top*(P)) has key k

Example: Search in Skip Lists

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



Insert in Skip Lists

Skip-Insert(S, k, v)

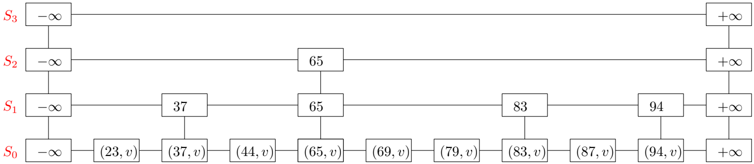
- Randomly repeatedly toss a coin until you get tails
- Let i the number of times the coin came up heads; this will be the height of the tower of k

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

- Increase height of skip list, if needed, to have $h > i$ levels.
- Search for k with *Skip-Search*(S, 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 $S_0, S_1, \cdots, S_i$
- Insert (k, v) after p_0 in S_0, and k after p_j in S_j for $1 \leq j \leq i$

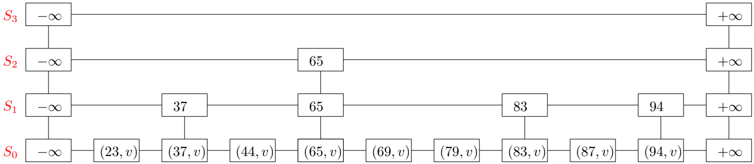
Example: Insert in Skip Lists

Example: Skip-Insert(S, 52, v)



Example 2: Insert in Skip Lists

Example: Skip-Insert$(S, 100, v)$



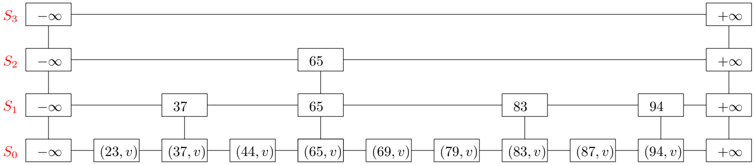
Delete in Skip Lists

Skip-Delete(S, k)

- Search for k with *Skip-Search*(S, k) to get stack P.
- P contains all predecessors $p_0, p_1, \ldots, p_h$ of k in lists $S_0, \ldots, S_h$.
- For each $0 \leq j \leq h$, if $key(after(p_j)) = k$, then remove $after(p_j)$ from list S_j
- Remove all but one of the lists S_i that contain only the two special keys

Example: 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 $1 - 1/n^2$
- Crucial for all operations:
  - How often do we **drop down** (execute $p \leftarrow \mathit{below}(p)$)?
  - How often do we **scan forward** (execute $p \leftarrow \mathit{after}(p)$)?
- *Skip-Search*: $O(\log n)$ expected time
  - $\#$ drop-downs $=$ height
  - expected $\#$ scan-forwards is ≤ 2 in each level
- *Skip-Insert*: $O(\log n)$ expected time
- *Skip-Delete*: $O(\log n)$ expected time
- Skip lists are fast and simple to implement in practice

Outline

1. Dictionaries with Lists revisited
   - Dictionary ADT: Implementations thus far
   - Skip Lists
   - Re-ordering Items

Re-ordering Items

- Recall: Unordered array implementation of ADT Dictionary
  search: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Arrays are a very simple and popular implementation. Can we do something to make search more effective in practice?
- No: if items are accessed equally likely
- Yes: otherwise (we have a probability distribution of the items)
  - Intuition: Frequently accessed items should be in the front.
  - Two cases: Do we know the access distribution beforehand or not?
  - For short lists or extremely unbalanced distributions this may be faster than AVL trees or Skip Lists, and much easier to implement.

Optimal Static Ordering

Example:

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

- 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 has expected access cost
  $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54$

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

- What if we do *not know the access probabilities* ahead of time?
- Rule of thumb (*temporal locality*): A recently accessed item is likely to be used soon again.
- Always insert at the front.
- **Move-To-Front** (MTF): Upon a successful search, move the accessed item to the front of the list

A	B	C	D	E			

$\downarrow$ Search(D)

D	A	B	C	E			

$\downarrow$ Insert(F)

F	D	A	B	C	E		

Dynamic Ordering: Transpose

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

A	B	C	D	E			

$\downarrow$ Search(D)

A	B	D	C	E			

$\downarrow$ Insert(F)

F	A	B	D	C	E		

Performance of dynamic ordering:

- Both can be implemented in arrays or linked lists.
- Transpose does not adapt quickly to changing access patterns.
- MTF Works well in practice.
- **Can show:** MTF is "2-competitive":
  No more than twice as bad as the optimal "offline" ordering.