CS 240 – Data Structures and Data Management

Module 5: Other Dictionary Implementations

A. Jamshidpey G. Kamath É. Schost
Based on lecture notes by many previous cs240 instructors

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

Spring 2020

References: Sedgewick 9.1-9.4

version 2020−06−15 23:29

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 we have seen so far:

- **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 BST** (AVL trees):
  $\Theta(\log n)$ search, insert, and delete

Remarks

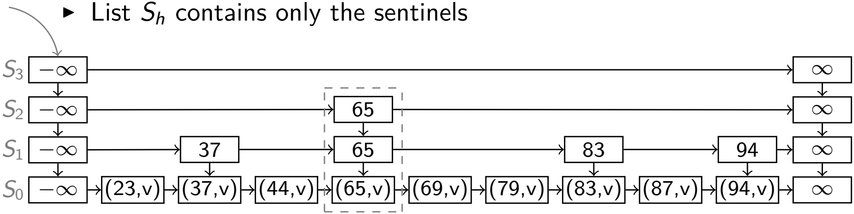
- Can show: the average-case height of binary search trees (over all possible insertion sequences) is $O(\log n)$.
- We can shift the average-case to expected height via randomization.

Outline

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

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



- Each KVP belongs to a **tower** of nodes
- There are (usually) more *nodes* than *keys*
- The skip list consists of a reference to the topmost left node.
- Each node p has references *p.after* and *p.below*

Search in Skip Lists

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

getPredecessors (k)

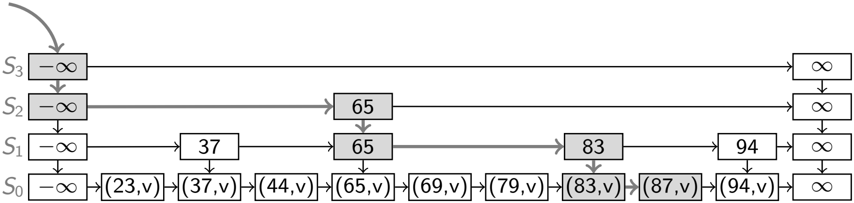
$p \leftarrow$ topmost left sentinel
$P \leftarrow$ stack of nodes, initially containing p
while $p.below \neq \texttt{NIL}$ **do**
    $p \leftarrow p.below$
    while $p.after.key < k$ **do** $p \leftarrow p.after$
    $P.push(p)$
return P

skipList::search (k)

$P \leftarrow$ *getPredecessors*(k)
$p_0 \leftarrow P.top()$ // predecessor of k in S_0
if $p_0.after.key = k$ **return** $p_0.after$
else return "not found, but would be after p_0"

Example: Search in Skip Lists

Example: *search*(87)



Insert in Skip Lists

skipList::insert(k, v)

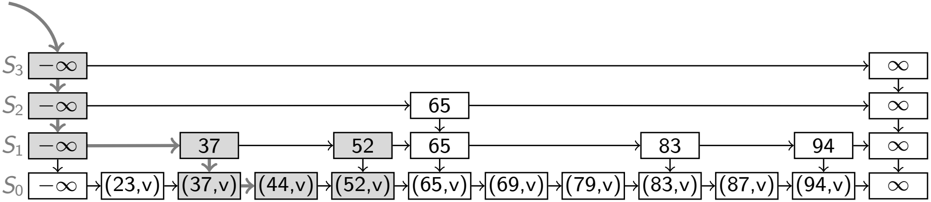
- Randomly repeatedly toss a coin until you get tails
- Let i the number of times the coin came up heads; this will be the height of the tower of k

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

- Increase height of skip list, if needed, to have $h > i$ levels.
- Use *getPredecessors*(k) to get stack P.
  The top $i+1$ 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: *skipList::insert*$(52, v)$
Coin tosses: H,T $\Rightarrow i = 1$
getPredecessors(52)



Example 2: Insert in Skip Lists

Example: $\textit{skipList::insert}(100, v)$

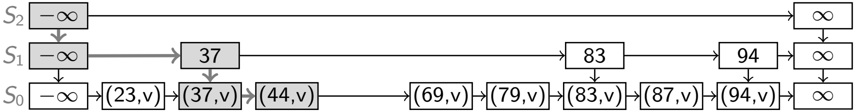
Delete in Skip Lists

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

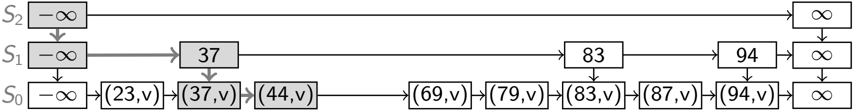
```
skipList::delete(k)
    P ← getPredecessors(k)
    while P is non-empty
        p ← P.pop()          // predecessor of k in some layer
        if p.after.key = k
            p.after ← p.after.after
        else break           // no more copies of k

    p ← topmost left sentinel
    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



skipList::delete(k)

...

$p \leftarrow$ topmost left sentinel

while $p.below.after$ is the ∞-sentinel

// the two top lists are both only sentinels, remove one

$p.below \leftarrow p.below.below$

$p.after.below \leftarrow p.after.below.below$

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 $p \leftarrow p.below$)?
  - How often do we *scan forward* (execute $p \leftarrow p.after$)?
- *skipList::search*: $O(\log n)$ expected time
  - $\#$ drop-downs $=$ height
  - expected $\#$ scan-forwards is ≤ 2 in each level
- *skipList::insert*: $O(\log n)$ expected time
- *skipList::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 list implementation of ADT Dictionary
  search: $\Theta(n)$, *insert*: $\Theta(1)$, *delete*: $\Theta(1)$ (after a search)
- Linked lists 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:

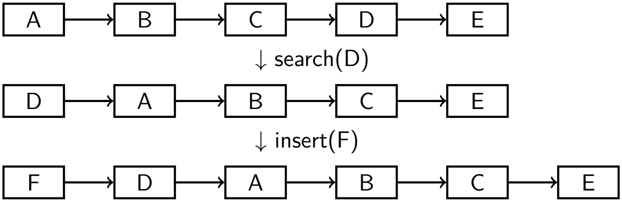
key	A	B	C	D	E
number of times we access it	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- We count cost i for accessing the key in the ith position.
- 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
  $\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{54}{26} \approx 2.07$

- **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

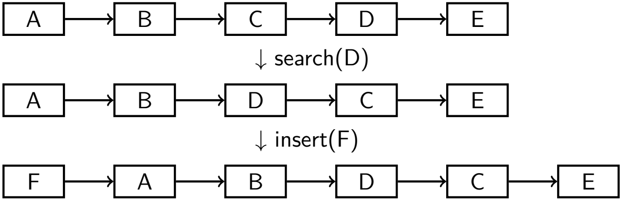
- What if we do *not know the access probabilities* ahead of time?
- Rule of thumb (**temporal locality**): A recently accessed item is likely to be used soon again.
- In list: Always insert at the front
- **Move-To-Front heuristic** (MTF): Upon a successful search, move the accessed item to the front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

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



Performance of dynamic ordering:

- 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 static ordering.