Module 5: Other Dictionary Implementations

T. Biedl   É. Schost   O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2021
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**: $\Theta(\text{height})$ search, insert and delete
- **Balanced BST** (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 linked list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Binary search trees**: $\Theta(\text{height})$ search, insert and delete
- **Balanced BST** (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

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; the left sentinel is the root

![Diagram of skip lists with levels $S_0$ through $S_3$ and nodes with keys and values]
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 left sentinel is the root

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/deletion*.

\[
\text{getPredecessors} \ (k) \\
1. \quad p \leftarrow \text{root} \\
2. \quad P \leftarrow \text{stack of nodes, initially containing } p \\
3. \quad \textbf{while } p.\text{below} \neq \text{NIL} \textbf{ do} \\
4. \quad \quad p \leftarrow p.\text{below} \\
5. \quad \quad \textbf{while } p.\text{after}.\text{key} < k \textbf{ do } p \leftarrow p.\text{after} \\
6. \quad \quad P.\text{push}(p) \\
7. \quad \textbf{return } P
\]

\[
\text{skipList::search} \ (k) \\
1. \quad P \leftarrow \text{getPredecessors}(k) \\
2. \quad p_0 \leftarrow P.\text{top}() \ // \ \text{predecessor of } k \ \text{in } S_0 \\
3. \quad \textbf{if } p_0.\text{after}.\text{key} = k \quad \textbf{return } p_0.\text{after} \\
4. \quad \textbf{else return } \text{“not found, but would be after } p_0\text{”}
\]
Example: Search in Skip Lists

Example: $\text{search}(87)$
Example: Search in Skip Lists

**Example**: \(\text{search}(87)\)

\[
\begin{align*}
S_3 & \rightarrow \infty \\
S_2 & \rightarrow \infty & \rightarrow 65 & \rightarrow \infty \\
S_1 & \rightarrow \infty & \rightarrow 37 & \rightarrow 65 & \rightarrow 83 & \rightarrow 94 & \rightarrow \infty \\
S_0 & \rightarrow \infty & \rightarrow (23,v) & \rightarrow (37,v) & \rightarrow (44,v) & \rightarrow (65,v) & \rightarrow (79,v) & \rightarrow (83,v) & \rightarrow (87,v) & \rightarrow (94,v) & \rightarrow \infty
\end{align*}
\]
Example: Search in Skip Lists

**Example:** $search(87)$
Example: Search in Skip Lists

Example: \textit{search}(87)

\begin{itemize}
  \item \textcolor{blue}{key compared with } \textcolor{red}{k} \textcolor{blue}{\downarrow}
  \item \textcolor{blue}{added to } \textcolor{red}{P} \textcolor{blue}{\downarrow}
\end{itemize}

\begin{itemize}
  \item \textcolor{blue}{key compared with } \textcolor{red}{k} \textcolor{blue}{\downarrow}
  \item \textcolor{blue}{added to } \textcolor{red}{P} \textcolor{blue}{\downarrow}
\end{itemize}
Example: Search in Skip Lists

Example: \textit{search}(87)

- $S_0: \infty \rightarrow (23,v) \rightarrow (37,v) \rightarrow (44,v) \rightarrow (65,v) \rightarrow (69,v) \rightarrow (79,v) \rightarrow (83,v) \rightarrow (87,v) \rightarrow (94,v) \rightarrow \infty$
- $S_1: \infty \rightarrow 37 \rightarrow 65 \rightarrow \infty$
- $S_2: \infty \rightarrow 65 \rightarrow \infty$
- $S_3: \infty \rightarrow \infty$

- key compared with $k$
- added to $P$
Example: Search in Skip Lists

Example: $\text{search}(87)$

- $S_0$: $-\infty$ \rightarrow (23,v) \rightarrow (37,v) \rightarrow (44,v) \rightarrow (65,v) \rightarrow (69,v) \rightarrow (79,v) \rightarrow (83,v) \rightarrow (87,v) \rightarrow (94,v) \rightarrow \infty$
- $S_1$: $-\infty$ \rightarrow 37 \rightarrow 65 \rightarrow 83 \rightarrow 94 \rightarrow \infty$
- $S_2$: $-\infty$ \rightarrow 65 \rightarrow \infty$
- $S_3$: $-\infty$ \rightarrow \infty

- key compared with $k$
- added to $P$
Insert in Skip Lists

\texttt{skipList::insert}(k, v)

- Randomly repeatedly toss a coin until you get tails
- Let \( i \) the number of times the coin came up heads
  - we want \( k \) to be in lists \( S_0, \ldots, S_i \).
  - \( i \to \text{height} \) of 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 \texttt{getPredecessors}(k) to get stack \( P \).
  - The top \( i \) items of \( P \) are the predecessors \( p_0, p_1, \ldots, p_i \) of where \( k \) should be in each list \( S_0, S_1, \ldots, S_i \)
- 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 ⇒ i = 1
Example: Insert in Skip Lists

Example: $\text{skipList}::\text{insert}(52, v)$

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

$\text{getPredecessors}(52)$
Example: Insert in Skip Lists

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

Coin tosses: H, T ⇒ i = 1

`getPredecessors(52)`
Example 2: Insert in Skip Lists

Example: \texttt{skipList::insert(100, v)}
Coin tosses: H,H,H,T \implies i = 3
Example 2: Insert in Skip Lists

Example: \texttt{skipList::insert}(100, \nu)
Coin tosses: H,H,H,T \Rightarrow i = 3

\textit{Height increase}
Example 2: Insert in Skip Lists

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

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

\textit{Height increase}

\textit{getPredecessors}(100)
Example 2: Insert in Skip Lists

Example: `skipList::insert(100, v)`
Coin tosses: H, H, H, T ⇒ i = 3

*Height increase*

`getPredecessors(100)`
**Insert in Skip Lists**

```plaintext
skipList::insert(k, v)
1. \( P \leftarrow \text{getPredecessors}(k) \)
2. for \((i \leftarrow 0; \text{random}(2) = 1; i \leftarrow i+1) \) \{\} // random tower height
3. while \( i \geq P\.size() \) // increase skip-list height?
4. root \leftarrow \text{new sentinel-only list linked in appropriate}
5. \( P\.append(\text{left sentinel of root}) \)
6. \( p \leftarrow P\.pop() \) // insert \((k, v)\) in \(S_0\)
7. \( z_{below} \leftarrow \text{new node with } (k, v), \text{inserted after } p \)
8. while \( i > 0 \) // insert \(k\) in \(S_1, \ldots, S_i\)
9. \( p \leftarrow P\.pop() \)
10. \( z \leftarrow \text{new node with } k \text{ added after } p \)
11. \( z\.below \leftarrow z_{below}; z_{below} \leftarrow z \)
12. \( i \leftarrow i - 1 \)
```
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)
1. \( P \leftarrow getPredecessors(k) \)
2. while \( P \) is non-empty
3. \( p \leftarrow P.pop() \)       // predecessor of \( k \) in some layer
4. if \( p.after.key = k \)
5. \( p.after \leftarrow p.after.after \)
6. else break                   // no more copies of \( k \)
7. \( p \leftarrow \) left sentinel of the root-list
8. while \( p.below.after \) is the \( \infty \)-sentinel
   // the two top lists are both only sentinels, remove one
9. \( p.below \leftarrow p.below.below \)
10. \( p.after.below \leftarrow p.after.below.below \)
```
Example: Delete in Skip Lists

Example: \texttt{skipList::delete}(65)
Example: Delete in Skip Lists

Example: \texttt{skipList::delete}(65)
\texttt{getPredecessors}(65)
Example: Delete in Skip Lists

Example: \textit{skipList::delete}(65)
\textit{getPredecessors}(65)
Example: Delete in Skip Lists

Example: `skipList::delete(65)`

`getPredecessors(65)`

*Height decrease*
Analysis of Skip Lists

- **Expected space usage**: $O(n)$
- **Expected height**: $O(\log n)$
- 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 $\leq 1$ in each level
- `skipList::insert`: $O(\log n)$ expected time
- `skipList::delete`: $O(\log n)$ expected time
Summary of Skip Lists

- $O(n)$ expected space, all operations take $O(\log n)$ expected time.
- As described they are no faster than randomized binary search trees.
- Can show: A biased coin-flip to determine tower-height gives smaller expected run-times.
- Can save links (hence space) by implementing towers as array.

Then skip lists are fast in practice and simple to implement.
Outline

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

- Recall: Unordered list/array implementation of ADT Dictionary
  - search: \(\Theta(n)\), insert: \(\Theta(1)\), delete: \(\Theta(1)\) (after a search)

- Lists/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.
Re-ordering Items

- Recall: Unordered list/array implementation of ADT Dictionary
  - search: $\Theta(n)$, insert: $\Theta(1)$, delete: $\Theta(1)$ (after a search)
- Lists/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:

<table>
<thead>
<tr>
<th>key</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency of access</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>access-probability</td>
<td>(\frac{2}{26})</td>
<td>(\frac{8}{26})</td>
<td>(\frac{1}{26})</td>
<td>(\frac{10}{26})</td>
<td>(\frac{5}{26})</td>
</tr>
</tbody>
</table>

- We count cost \(i\) for accessing the key in the \(i\)th 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
  \[
  \frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54
  \]
Optimal Static Ordering

Example:

<table>
<thead>
<tr>
<th>key</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency of access</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>access-probability</td>
<td>(\frac{2}{26})</td>
<td>(\frac{8}{26})</td>
<td>(\frac{1}{26})</td>
<td>(\frac{10}{26})</td>
<td>(\frac{5}{26})</td>
</tr>
</tbody>
</table>

- We count cost \(i\) for accessing the key in the \(i\)th 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
  \[
  \frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54
  \]

- **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.
- 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

![Diagram of MTF sequence]

- 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.

- **Initial order**: A → B → C → D → E
  - **Search(D)**: A → B → D → C → E
  - **Insert(F)**: F → A → B → D → C → E

**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.
Dynamic Ordering: Transpose

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

```
A → B → C → D → E
↓ search(D)
A → B → D → C → E
```

```
F → A → B → D → C → E
↓ insert(F)
```

**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.