Module 5: Other Dictionary Implementations

Mark Petrick
Based on lecture notes by many previous cs240 instructors

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

Fall 2020

References: Sedgewick 9.1-9.4
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 (\textit{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 \textit{root}

![Diagram of skip lists]

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 \( p.below \).
Skip Lists

- A hierarchy $S$ of ordered linked lists (\textit{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 \textit{root}

- Each KVP belongs to a \textit{tower} of nodes
- There are (usually) more \textit{nodes} than \textit{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*.

```plaintext
getPredecessors (k)
1. \( p \leftarrow \) topmost left sentinel
2. \( P \leftarrow \) stack of nodes, initially containing \( p \)
3. while \( p.below \neq \) NIL do
4. \( p \leftarrow p.below \)
5. while \( p.after.key < k \) do \( p \leftarrow p.after \)
6. \( P.push(p) \)
7. \( \text{return } P \)
```

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

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

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

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

Example: \textit{search}(87)

- \( S_3 \):
  - key compared with 87
  - added to \( P \)

- \( S_2 \):
  - 65

- \( S_1 \):
  - key compared with 37

- \( S_0 \):
  - (23, \( v \))
  - (37, \( v \))
  - (44, \( v \))
  - (65, \( v \))
  - (69, \( v \))
  - (79, \( v \))
  - (83, \( v \))
  - (87, \( v \))
  - (94, \( v \))

- End of list: \( \infty \)
Example: Search in Skip Lists

Example: \textit{search}(87)

\hspace{1cm}

\textcolor{blue}{\textbf{key compared with } k}

\textcolor{violet}{\textbf{added to } P}
Example: Search in Skip Lists

Example: search(87)

key compared with \( k \)
added to \( P \)
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
  - we want $k$ to be in lists $S_0, \ldots, S_i$.
  - $i \rightarrow \text{height}$ of tower of $k$
  - $P($tower of key $k$ has height $\geq i) = (\frac{1}{2})^i$
- Increase height of skip list, if needed, to have $h > i$ levels.
- Use `getPredecessors(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: \texttt{skipList::insert}(52, v)
Coin tosses: H,T \Rightarrow i = 1
Example: Insert in Skip Lists

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

Coin tosses: H,T ⇒ i = 1

*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: $\textit{skipList}::\text{insert}(100, v)$
Coin tosses: H,H,H,T $\Rightarrow i = 3$
Example 2: Insert in Skip Lists

Example: \textit{skipList::insert}(100, v)
Coin tosses: H,H,H,T ⇒ i = 3

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

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

Coin tosses: H, H, H, T ⇒ \( i = 3 \)

\textit{Height increase}

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

Example: \textit{skipList::insert}(100, v)
Coin tosses: H,H,H,T ⇒ i = 3

\textit{Height increase}

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

\[ \text{skipList::insert}(k, v) \]

1.  \( P \leftarrow \text{getPredecessors}(k) \)
2.  \( \text{for } (i \leftarrow 0; \text{random}(2) = 1; i \leftarrow i + 1) \{} \) \hspace{1cm} // random tower height
3.  \( \text{while } i \geq P.\text{size()} \) \hspace{1cm} // increase skip-list height?
4.  \( \text{root} \leftarrow \text{new sentinel-only list,} \)
    \( \quad \text{linked to previous root-list appropriately} \)
5.  \( P.\text{append}(\text{left sentinel of root}) \)
6.  \( p \leftarrow P.\text{pop()} \) \hspace{1cm} // insert \((k, v)\) in \(S_0\)
7.  \( k_{below} \leftarrow \text{new node with } (k, v), \text{inserted after } p \)
8.  \( \text{while } i > 0 \) \hspace{1cm} // insert \(k\) in \(S_1, \ldots, S_i\)
9.  \( p \leftarrow P.\text{pop()} \)
10. \( k_{below} \leftarrow \text{new node with } k, \)
    \( \quad \text{inserted after } p \text{ with below-reference to } k_{below} \)
11. \( i \leftarrow i - 1 \)
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. \( P \leftarrow \text{getPredecessors}(k) \)
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: `skipList::delete(65)`
Example: Delete in Skip Lists

Example: `skipList::delete(65)`

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

Example: \textit{skipList::delete}(65)

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

Example: \texttt{skipList::delete}(65)  
\texttt{getPredecessors}(65)  
\textit{Height decrease}
Analysis 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 $\leq 1$ 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
  $$\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54$$

Claim: Overall 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.
**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

```
| A | B | C | D | E |
```

↓ search(D)

```
| D | A | B | C | E |
```

↓ insert(F)

```
| F | D | A | B | C | E |
```

- 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.

- **Before search(D)**: A → B → C → D → E
- **After search(D)**: A → B → D → C → E
  - ↓ search(D)
- **Before insert(F)**: F → A → B → D → C → E
- **After insert(F)**: F → A → B → D → C → E
  - ↓ insert(F)
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  
↓ 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.