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*): $\Theta(\log n)$ search, insert, and delete
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.
Optimal Static Ordering

A list of elements ordered by non-increasing probability of access has minimum expected access cost

- \( L = \langle x_1, x_2, \ldots, x_n \rangle \)

Expected access cost in \( L \) is
\[
E(L) = \sum_{i=1}^{n} P(x_i) \cdot T(x_i) = \sum_{i=1}^{n} P(x_i) \cdot i
\]

- \( P(x_i) \) - access probability for \( x_i \)
- \( T(x_i) \) - position of \( x_i \) in \( L \)

Example
\[
P(a) = 0.3 \quad P(b) = 0.5 \quad P(c) = 0.2
\]
\[
L = \langle a, b, c \rangle
\]
\[
E(L) = 0.3 + 0.5 \cdot 2 + 0.2 \cdot 3 = 1.9
\]
\[
L = \langle b, a, c \rangle
\]
\[
E(L) = 0.5 + 0.3 \cdot 2 + 0.2 \cdot 3 = 1.7
\]
Optimal Static Ordering

A list of elements ordered by non-increasing probability of access has minimum expected access cost

Proof by Contradiction

- \( L = \langle x_1, \ldots, x_k, x_{k+1}, \ldots, x_n \rangle \)
  
  Suppose the access cost of \( L \) is optimal and there is \( k \) such that \( P(x_k) < P(x_{k+1}) \)

  \[
  E(L) = P(x_k) \cdot k + P(x_{k+1}) \cdot (k + 1) + \sum_{i \neq k, k+1} P(x_i) \cdot i
  \]

- Create another list \( L' \) by swapping \( x_k \) and \( x_{k+1} \).
  
  \( L' = \langle x_1, \ldots, x_{k+1}, x_k, \ldots, x_n \rangle \)

  \[
  E(L') = P(x_{k+1}) \cdot k + P(x_k) \cdot (k + 1) + \sum_{i \neq k, k+1} P(x_i) \cdot i
  \]

- \( E(L') - E(L) = P(x_k) - P(x_{k+1}) < 0 \Rightarrow E(L') < E(L) \)

  Contradiction
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 does not adapt quickly to changing access patterns.
- MTF works well in practice.
- Theoretically MTF is “competitive”: No more than twice as bad as the optimal “offline” ordering.
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 does not adapt quickly to changing access patterns.
- 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

![Diagram of skip lists](attachment:image.png)
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

\(\text{skip-search}(L, k)\)
\(L: A \text{ skip list, } k: \text{ a key}\)

1. \(p \leftarrow \text{topmost left position of } L\)
2. \(S \leftarrow \text{stack of positions, initially containing } p\)
3. \(\text{while below}(p) \neq \text{null do}\)
4. \(p \leftarrow \text{below}(p)\)
5. \(\text{while key(after}(p)) < k \text{ do}\)
6. \(p \leftarrow \text{after}(p)\)
7. \(\text{push } p \text{ onto } S\)
8. \(\text{return } S\)

- \(S\) contains positions of the largest key \textbf{less than } \(k\) at each level.
- \(\text{after(top}(S))\) will have key \(k\), iff \(k\) is in \(L\).
- \textbf{drop down: } \(p \leftarrow \text{below}(p)\)
- \textbf{scan forward: } \(p \leftarrow \text{after}(p)\)
Search in Skip Lists

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{skip_list_example}
\end{figure}
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 \(i\) be the number of times the coin came up heads
  - Search for \(k\) in the skip list and find the positions \(p_0, p_1, \ldots, p_i\) of the items with largest key less than \(k\) in each list \(S_0, S_1, \ldots, S_i\) (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, ν)
Coin tosses: H, T ⇒ 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, ν)
Coin tosses: H, T ⇒ i = 1
Insert in Skip Lists

Example: Skip-Insert$(S, 100, \nu)$
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 ⇒ i = 3

$\text{Skip-Search}(S, 100)$
Insert in Skip Lists

Example: Skip-Insert(\(S, 100, v\))
Coin tosses: H,H,H,T \(\implies i = 3\)
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 \(\text{key}(\text{after}(p_i)) == k\), then remove \(\text{after}(p_i)\) 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