CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

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
Direct Addressing

**Special situation:** For a known \( M \in \mathbb{N} \), every key \( k \) is an integer with \( 0 \leq k < M \).

We can then implement a dictionary easily: Use an array \( A \) of size \( M \) that stores \((k, \nu)\) via \( A[k] \leftarrow \nu \).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dog</td>
<td></td>
<td>cat</td>
<td>pig</td>
</tr>
</tbody>
</table>

- **search**\((k)\): Check whether \( A[k] \) is NIL
- **insert**\((k, \nu)\): \( A[k] \leftarrow \nu \)
- **delete**\((k)\): \( A[k] \leftarrow \text{NIL} \)
Direct Addressing

**Special situation:** For a known \( M \in \mathbb{N} \), every key \( k \) is an integer with \( 0 \leq k < M \).

We can then implement a dictionary easily: Use an array \( A \) of size \( M \) that stores \((k, v)\) via \( A[k] \leftarrow v \).

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>cat</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>pig</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **search**\((k)\): Check whether \( A[k] \) is NIL
- **insert**\((k, v)\): \( A[k] \leftarrow v \)
- **delete**\((k)\): \( A[k] \leftarrow \text{NIL} \)

Each operation is \( \Theta(1) \).
Total space is \( \Theta(M) \).

What sorting algorithm does this remind you of? *Bucket Sort*
Hashing

Two disadvantages of direct addressing:

- It cannot be used if the keys are not integers.
- It wastes space if $M$ is unknown or $n \ll M$.

**Hashing idea:** Map (arbitrary) keys to integers in range $\{0, \ldots, M-1\}$ and then use direct addressing.

Details:

- **Assumption:** We know that all keys come from some universe $U$.
  (Typically $U = \mathbb{N}$.)
- We design a hash function $h : U \rightarrow \{0, 1, \ldots, M - 1\}$.
  (Commonly used: $h(k) = k \bmod M$. We will see other choices later.)
- Store dictionary in hash table, i.e., an array $T$ of size $M$.
- An item with key $k$ should ideally be stored in slot $h(k)$, i.e., at $T[h(k)]$. 
Hashing example

\[ U = \mathbb{N}, \ M = 11, \ h(k) = k \mod 11. \]

The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).

\[
\begin{array}{c|c}
0 & \text{ } \\
1 & 45 \\
2 & 13 \\
3 & \text{ } \\
4 & 92 \\
5 & 49 \\
6 & \text{ } \\
7 & 7 \\
8 & \text{ } \\
9 & \text{ } \\
10 & 43 \\
\end{array}
\]

\[
45 = 44 + 1
\]

\[
= 4 \times 11 + 1
\]

\[
\Rightarrow 45 \mod 11 = 1
\]

\[
\Rightarrow h(45) = 1
\]
Collisions

- Generally hash function $h$ is not injective, so many keys can map to the same integer.
  - For example, $h(46) = 2 = h(13)$ if $h(k) = k \mod 11$.
- We get **collisions**: we want to insert $(k, v)$ into the table, but $T[h(k)]$ is already occupied.
- There are many strategies to resolve collisions:

  ![Diagram of hash collision resolution strategies]

  - **Chaining**
    - Multiple items at location
    - Many alternate slots (Probe sequence)
      - Linear Probing
    - Other probes...
  - **Open addressing**
    - Alternate slots in array
    - One alternate slot (Cuckoo Hashing)
    - Double Hashing
Probability of having a collision?

Suppose we pick \( n \) values in \( 0 \ldots H-1 \), independently, uniform distribution.

\[
\Pr(\text{no collision}) = \frac{H(H-1)(H-2)\ldots(H-(n-1))}{H^n}
\]

\[
= 1 \cdot \left(1 - \frac{1}{H}\right) \left(1 - \frac{2}{H}\right)\ldots \left(1 - \frac{n-1}{H}\right)
\]

\[
\Pr(\text{collision}) = 1 - \left(1 - \frac{1}{H}\right)\ldots \left(1 - \frac{n-1}{H}\right) = f(n, H)
\]

\[
f(22, 365) < 0.5 \quad f(23, 365) > 0.5
\]
Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies
Separate Chaining

Simplest collision-resolution strategy: Each slot stores a **bucket** containing 0 or more KVPs.

- A bucket could be implemented by any dictionary realization (even another hash table!).
- The simplest approach is to use unsorted linked lists for buckets. This is called collision resolution by **separate chaining**.

- **search**($k$): Look for key $k$ in the list at $T[h(k)]$. Apply MTF-heuristic!
- **insert**($k, v$): Add $(k, v)$ to the front of the list at $T[h(k)]$.
- **delete**($k$): Perform a search, then delete from the linked list.
Chaining example

\[ M = 11, \quad h(k) = k \mod 11 \]

*insert*(79)

\[ h(79) = 2 \]
Complexity of chaining

**Run-times:** *insert* takes time $O(1)$.

*search* and *delete* have run-time $O(1 + \text{size of bucket } T(h(k)))$.

- The *average* bucket-size is $\frac{\theta}{\alpha} =: \alpha$.
  ($\alpha$ is also called the *load factor*.)

- However, this does not imply that the *average-case* cost of *search* and *delete* is $O(1 + \alpha)$.
  (If all keys hash to the same slot, then the average bucket-size is still $\alpha$, but the operations take time $\Theta(n)$ on average.)

- **Uniform Hashing Assumption:** for any key $k$, and for any $j \in \{0, \ldots, M - 1\}$, $h(k) = j$ happens with probability $1/M$, independently of where the other keys hash to.
  (This depends on the input and how we choose the function $\Rightarrow$ later.)

- Under this assumption, each key is expected to collide with $\frac{n-1}{M}$ other keys and the average-case cost of *search* and *delete* is hence $O(1 + \alpha)$.  

We assume that computing $W(k)$ is constant time.
We hash keys $k_1, ..., k_m$. We assume

1) $\forall j \in \{0, ..., m-1\}$
   $\forall i \in \{1, ..., n\}$ $\text{Prob}(h(k_i) = j) = \frac{1}{m}$

2) $\forall i, j', \in \{0, ..., m-1\}$
   $\forall i, i' \in \{1, ..., n\}$, $i \neq i'$
   $$\text{Prob}(h(k_i) = j \text{ and } h(k_{i'}) = j') = \text{Prob}(h(k_i) = j) \cdot \text{Prob}(h(k_{i'}) = j') = \frac{1}{m^2}.$$
\( i \neq i' \)

\[
\text{Prob}(h(k) = h(k')) = \sum_{j=0}^{\frac{m-1}{n^2}} \frac{\text{Prob}(\ n(k) = h(k) = i \ )}{\frac{m-1}{n^2}}
\]

\( = \frac{1}{n}. \)

Let \( X_{ii'} = \begin{cases} 0 & \text{if } \ h(k) \neq h(k') \\ 1 & \text{if } \ h(k) = h(k') \end{cases} \)

\( \Rightarrow \) # collisions involving \( h_i = \sum_{i \neq i'} X_{ii'} \)

\( \Rightarrow E(\#..) = \sum_{i \neq i'} E(X_{ii'}) = \sum_{i \neq i'} \frac{1}{n} = \frac{n-1}{n} \)
Load factor and re-hashing

- For all collision resolution strategies, the run-time evaluation is done in terms of the load factor $\alpha = n/M$.

- We keep the load factor small by rehashing when needed:
  - Keep track of $n$ and $M$ throughout operations
  - If $\alpha$ gets too large, create new (twice as big) hash-table, new hash-function(s) and re-insert all items in the new table.

- Rehashing costs $\Theta(M + n)$ but happens rarely enough that we can ignore this term when amortizing over all operations.

- We should also re-hash when $\alpha$ gets too small, so that $M \in \Theta(n)$ throughout, and the space is always $\Theta(n)$.

Summary: If we maintain $\alpha \in \Theta(1)$, then (under the uniform hashing assumption) the average cost for hashing with chaining is $O(1)$ and the space is $\Theta(n)$. 

$$\frac{1}{4} \leq \alpha \leq 2 \quad \frac{1}{4} \leq \frac{n}{M} \leq 2$$

$$\frac{n}{2} \leq M \leq 4n \quad M \in \Theta(n)$$
Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies
Open addressing

**Main idea:** Avoid the links needed for chaining by permitting only one item per slot, but allowing a key $k$ to be in multiple slots.

*search* and *insert* follow a **probe sequence** of possible locations for key $k$: $\langle h(k, 0), h(k, 1), h(k, 2), \ldots \rangle$ until an empty spot is found.

*delete* becomes problematic:

- Cannot leave an empty spot behind; the next search might otherwise not go far enough.
- **lazy deletion:** Mark spot as *deleted* (rather than NIL) and continue searching past deleted spots.
Open addressing

**Main idea:** Avoid the links needed for chaining by permitting only one item per slot, but allowing a key $k$ to be in multiple slots.

*search* and *insert* follow a **probe sequence** of possible locations for key $k$: $\langle h(k, 0), h(k, 1), h(k, 2), \ldots \rangle$ until an empty spot is found.

*delete* becomes problematic:

- Cannot leave an empty spot behind; the next search might otherwise not go far enough.
- **lazy deletion:** Mark spot as *deleted* (rather than NIL) and continue searching past deleted spots.

Simplest method for open addressing: **linear probing**

$h(k, i) = (h(k) + i) \mod M$, for some hash function $h$. 
Linear probing example

\[ M = 11, \quad h(k, i) = (h(k) + i) \mod 11. \]
Linear probing example

$$M = 11, \quad h(k, i) = (h(k) + i) \mod 11.$$
Linear probing example

\[ M = 11, \quad h(k, i) = (h(k) + i) \mod 11. \]

\[
\begin{array}{c|c}
0 & 20 \\
1 & 45 \\
2 & 13 \\
3 & \\
4 & 92 \\
5 & 49 \\
6 & \\
7 & 7 \\
8 & 41 \\
9 & 84 \\
10 & \text{deleted} \\
\end{array}
\]

\textit{delete(43)}

\[ h(43, 0) = 10 \]
Linear probing example

\[ M = 11, \quad h(k, i) = (h(k) + i) \mod 11. \]

\[
\begin{array}{c|c}
0 & 20 \\
1 & 45 \\
2 & 13 \\
3 & \text{search(20)} \\
4 & 92 \\
5 & 49 \\
6 & 7 \\
7 & 41 \\
8 & 84 \\
9 & \text{deleted} \\
10 & \end{array}
\]

\[ h(20, 2) = 0 \]

found
probe-sequence::insert\( (T, (k, v)) \)
1. \texttt{for } \( j = 0; j < M; j++ \) 
2. \texttt{if } \( T[h(k, j)] \) is NIL or "deleted"
3. \( T[h(k, j)] = (k, v) \)
4. \texttt{return } "success"
5. \texttt{return } "failure to insert" \hspace{1em} // need to re-hash

probe-sequence-search\( (T, k) \)
1. \texttt{for } \( j = 0; j < M; j++ \) 
2. \texttt{if } \( T[h(k, j)] \) is NIL
3. \texttt{return } "item not found"
4. \texttt{else if } \( T[h(k, j)] \) has key \( k \)
5. \texttt{return } \( T[h(k, j)] \)
6. \hspace{1em} // ignore "deleted" and keep searching
7. \texttt{return } "item not found"
Independent hash functions

\[ \forall k \forall i \forall j \text{ in } 0, \ldots, n-1 \quad \text{Prob}(h_0(k) = i \text{ and } h_1(k) = j) = \text{Prob}(h_0(k) = i) \cdot \text{Prob}(h_1(k) = j) \]

- Some hashing methods require two hash functions \( h_0, h_1 \).
- These hash functions should be independent in the sense that the random variables \( P(h_0(k) = i) \) and \( P(h_1(k) = j) \) are independent.
- Using two modular hash-functions may often lead to dependencies.
- Better idea: Use multiplicative method for second hash function:
  \[ h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor, \]
  - \( A \) is some floating-point number
  - \( kA - \lfloor kA \rfloor \) computes fractional part of \( kA \), which is in \( [0, 1) \)
  - Multiply with \( M \) to get floating-point number in \( [0, M) \)
  - Round down to get integer in \( \{0, \ldots, M - 1\} \)

Knuth suggests \( A = \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618. \)
$A = 0.5 \quad M = 100 \quad h(i), i = 0 \ldots 19$
\[ A = \frac{1}{3} \quad H = 100 \quad h(i), \quad i = 0 \ldots 19 \]
\[A = \frac{\sqrt{5} - 1}{2}, \quad M = 100, \quad h(i), \quad i = 0 \ldots 19\]
Double Hashing

- Assume we have two hash independent functions $h_0, h_1$.
- Assume further that $h_1(k) \neq 0$ and that $h_1(k)$ is relative prime with the table-size $M$ for all keys $k$.
  - Choose $M$ prime.
  - Modify standard hash-functions to ensure $h_1(k) \neq 0$
    - E.g. modified multiplication method: $h(k) = 1 + [(M - 1)(kA - \lfloor kA \rfloor)]$

- **Double hashing**: open addressing with probe sequence

  $$h(k, i) = h_0(k) + i \cdot h_1(k) \mod M$$

- **search, insert, delete** work just like for linear probing, but with this different probe sequence.
Double hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \left\lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor + 1 \]

\text{insert}(41)

\[ h_0(41) = 8 \]
\[ h(41, 0) = 8 \]
Double hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \left\lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor + 1 \]

**insert(194)**

<table>
<thead>
<tr>
<th>( h_0(194) = 7 )</th>
<th>( h(194, 0) = 7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td><strong>194</strong></td>
</tr>
<tr>
<td>4</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
$M = 16$

$h$ such that $h_0(h) = 3$

$h_1(h) = 4$
Outline

1 Dictionaries via Hashing
   • Hashing Introduction
   • Separate Chaining
   • Probe Sequences
   • Cuckoo hashing
   • Hash Function Strategies
Cuckoo hashing

We use two independent hash functions $h_0, h_1$ and two tables $T_0, T_1$.

**Main idea:** An item with key $k$ can *only* be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$.

- *search* and *delete* then take constant time.
- *insert always* initially puts a new item into $T_0[h_0(k)]$.
  
  If $T_0[h_0(k)]$ is occupied: “kick out” the other item, which we then attempt to re-insert into its alternate position $T_1[h_1(k)]$.
  
  This may lead to a loop of “kicking out”. We detect this by aborting after too many attempts.
  
  In case of failure: rehash with a larger $M$ and new hash functions.

*insert* may be slow, but is expected to be constant time if the load factor is small enough.
cuckoo::insert\((k, v)\)
1. \(i \leftarrow 0\)
2. \(\text{do at most } 2n \text{ times:}\)
3. \(\text{if } T_i[h_i(k)] \text{ is NIL}\)
4. \(T_i[h_i(k)] \leftarrow (k, v)\)
5. \(\text{return } \text{“success”}\)
6. \(\text{swap}((k, v), T_i[h_i(k)])\)
7. \(i \leftarrow 1 - i\)
8. \(\text{return } \text{“failure to insert”} \quad // \text{need to re-hash}\)

After \(2n\) iterations, there definitely was a loop in the “kicking out” sequence (why?)

In practice, one would stop the iterations much earlier already.
Cuckoo hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \left\lfloor 11(\phi k - \lfloor \phi k \rfloor) \right\rfloor \]

**insert(95)**

\[ i = 0 \]
\[ k = 95 \]
\[ h_0(k) = 7 \]
\[ h_1(k) = 7 \]
Cuckoo hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = [11(\varphi k - \lfloor \varphi k \rfloor)] \]

**insert(95)**

\( i = 1 \)
\( k = 51 \)

\( h_0(k) = 7 \)
\( h_1(k) = 5 \)
Cuckoo hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert(26)**

- \( i = 0 \)
- \( k = 26 \)
- \( h_0(k) = 4 \)
- \( h_1(k) = 0 \)
Cuckoo hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert(26)**

\[ i = 0 \]
\[ k = 51 \]
\[ h_0(k) = 7 \]
\[ h_1(k) = 5 \]
Cuckoo hashing example

\[ M = 11, \quad h_0(k) = k \mod 11, \quad h_1(k) = \left\lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor \]

search(59)

\[ h_0(59) = 7 \]
\[ h_1(59) = 5 \]
Claim: if we enter the insert loop 2n+1 times, we don't exit the loop.

Proof: we have n keys k_1...k_n

2 tables

\rightarrow \exists k_i such that we insert k_i 3 times.

\rightarrow this will create an infinite loop.
Cuckoo hashing discussions

- The two hash-tables need not be of the same size.
- Load factor \( \alpha = \frac{n}{\text{size of } T_0 + \text{size of } T_1} = \frac{n}{2M} \)
- One can argue: If the load factor \( \alpha \) is small enough then insertion has \( O(1) \) expected run-time.
- This crucially requires \( \alpha < \frac{1}{2} \).

There are many possible variations:

- The two hash-tables could be combined into one.
- Be more flexible when inserting: Always consider both possible positions.
- Use \( k > 2 \) allowed locations (i.e., \( k \) hash-functions).
Complexity of open addressing strategies

For any open addressing scheme, we must have $\alpha < 1$ (why?). Cuckoo hashing requires $\alpha < 1/2$.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Avg.-case costs:} & \text{search (unsuccessful)} & \text{insert} & \text{search (successful)} \\
\hline
\text{Linear Probing} & \frac{1}{(1 - \alpha)^2} & \frac{1}{(1 - \alpha)^2} & \frac{1}{1 - \alpha} \\
\hline
\text{Double Hashing} & \frac{1}{1 - \alpha} & \frac{1}{1 - \alpha} & \frac{1}{\alpha \log\left(\frac{1}{1 - \alpha}\right)} \\
\hline
\text{Cuckoo Hashing} & \frac{1}{\alpha} & \frac{\alpha}{(1 - 2\alpha)^2} & \frac{1}{\alpha} \\
\hline
\end{array}
\]

\[\alpha \text{ is fixed, } \lim_{n \to \infty} \]

Summary: All operations have $O(1)$ average-case run-time if the hash-function is uniform and $\alpha$ is kept sufficiently small. But worst-case run-time is (usually) $\Theta(n)$. 
Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies
Choosing a good hash function

- **Goal:** Satisfy uniform hashing assumption (each hash-index is equally likely)

- Proving this is usually impossible, as it requires knowledge of the input distribution and the hash function distribution.

- We can get good performance by choosing a hash-function that is
  - unrelated to any possible patterns in the data, and
  - depends on all parts of the key.

- We saw two basic methods for integer keys:
  - **Modular method:** $h(k) = k \mod M$.
    We should choose $M$ to be a prime.
  - **Multiplicative method:** $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
    for some constant floating-point number $A$ with $0 < A < 1$. 

  


Universal Hashing

Every hash function must do badly for some sequences of inputs:

- If the universe contains at least \( \frac{M}{n} \) keys, then there are \( n \) keys that all hash to the same value.
  \[
  n = 3, \quad M = 2, \quad n(n-1)/2 = 5
  \]
- For this set of keys, we have the worst case.

**Idea:** Randomization!

- When initializing or re-hashing, use as hash function
  \[
  h(k) = ((ak + b) \mod p) \mod M
  \]
  where \( p > M \) is a prime number, and \( a, b \) are random numbers in \( \{0, \ldots, p - 1\} \), \( a \neq 0 \).
- Can prove: For any (fixed) numbers \( x \neq y \), the probability of a collision using this random function \( h \) is at most \( \frac{1}{M} \).
- Therefore the expected run-time is \( O(1) \) if \( \alpha \) is kept small enough.

We have again shifted the performance from "bad input" to "bad luck".
Multi-dimensional Data

What if the keys are multi-dimensional, such as strings in $\Sigma^*$?

Standard approach is to \textit{flatten} string $w$ to integer $f(w) \in \mathbb{N}$, e.g.

$$A \cdot P \cdot P \cdot L \cdot E \quad \rightarrow \quad \left(65, 80, 80, 76, 69\right) \quad \text{(ASCII)}$$

$$\rightarrow \quad 65R^4 + 80R^3 + 80R^2 + 76R^1 + 68R^0$$

(for some radix $R$, e.g. $R = 255$)

We combine this with a modular hash function: $h(w) = f(w) \mod M$

$H \text{ fixed, } R = 255$

To compute this in $O(|w|)$ time without overflow, use Horner's rule and apply mod early. For example, $h(APPLE)$ is

$$(\left(\left(\left(\left(65R + 80\right)R + 80\right)R + 76\right)R + 68\right) \mod M \quad \mod H)$$

$= \left(\left(\left(\left(\left(\left(\left(65R + 80\right) \mod M\right)R + 80\right) \mod M\right)R + 76\right) \mod M\right)R + 69\right) \mod M \quad \mod H$
Hashing vs. Balanced Search Trees

**Advantages of Balanced Search Trees**

- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- Predictable space usage (exactly $n$ nodes)
- Never need to rebuild the entire structure
- Supports ordered dictionary operations (rank, select etc.)

**Advantages of Hash Tables**

- $O(1)$ operations (if hashes well-spread and load factor small)
- We can choose space-time tradeoff via load factor
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete