Algorithms
(Hash Tables)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

September 19, 2021
Contents

- Hash Functions
- Hash Tables
- Collision-Avoiding Techniques
  - Separate Chaining
  - Open Addressing
- Applications
### Definition

- A hash function is a function that maps arbitrary size data to fixed size data.

<table>
<thead>
<tr>
<th>Arbitrary size data</th>
<th>Key</th>
<th>Hash function</th>
<th>Hash</th>
<th>Fixed size data</th>
</tr>
</thead>
</table>

---

**Hash function**
Cryptographic hash function

Input

- Fox
- The red fox jumps over the blue dog
- The red fox jumps over the blue dog
- The red fox jumps over the blue dog

Digest

- DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
- 0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
- 8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
- FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
- 8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

Properties of an ideal cryptographic hash function

- Deterministic and fast
- Computing message from hash value is infeasible
- Computing two messages having same hash value is infeasible
- Tiny change in message must change the hash value drastically
Applications of hashing

- Web page search using URLs
- Password verification
- Symbol tables in compilers
- Filename-filepath linking in operating systems
- Plagiarism detection using Rabin-Karp string matching algorithm
- English dictionary search
- Used as part of the following concepts:
  - finding distinct elements
  - counting frequencies of items
  - finding duplicates
  - message digests
  - commitment
  - Bloom filters
Password authentication

Username: Charles  
Password: Darwin

hashed password: Charles: HRFDNMD

Login

Database of encrypted passwords

Database of encrypted passwords

Charles: HRFDNMD
A hash table is a data structure to implement dictionary ADT.

A hash table is an efficient implementation of a set/multiset/map/multimap.

A hash table performs insert, delete, and search operations in constant expected time.
Balanced search trees vs. Hash tables

- Balanced search tree is ideal for sorted collection
- Hash table is ideal for unsorted collection

<table>
<thead>
<tr>
<th>Operations</th>
<th>Balanced tree (worst)</th>
<th>Hash table (avg.)</th>
<th>Hash table (worst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting-unrelated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>$O (\log n)$</td>
<td>$O (1)$</td>
<td>$O (n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O (\log n)$</td>
<td>$O (1)$</td>
<td>$O (n)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O (\log n)$</td>
<td>$O (1)$</td>
<td>$O (n)$</td>
</tr>
<tr>
<td>Sorting-related</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>$O (n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Predecessor</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Successor</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Range-Minimum</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Range-Maximum</td>
<td>$O (\log n)$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>Range-Sum</td>
<td>$O (n)$</td>
<td>$\times$</td>
<td></td>
</tr>
</tbody>
</table>
Hash table

Arbitrary key $k$

\begin{align*}
\text{Key space} & \\
\begin{array}{|c|}
\hline
0 \\
1 \\
2 \\
\vdots \\
\mathcal{H}(k) & (k, v) \\
\vdots \\
N - 1 \\
\hline
\end{array}
\end{align*}

Hash space
A hash function is a mapping from arbitrary objects to the set of indices $[0, N - 1]$.
The key-value pair $(k, v)$ is stored at $A[\mathcal{H}(k)]$ in the hash table.
# Hash table

## Questions

1. **How can keys of arbitrary objects be mapped to array indices that are whole numbers?**
2. **How can an infinite number of keys be mapped to a finite number of indices?**
3. **What are the pros/cons of hash tables w.r.t. balanced trees?**
4. **What are the properties of an ideal hash function?**
5. **Is there one practical hash function that is best for all input?**
6. **What is the hash function used in Java?**
7. **Will there be collisions during hashing?**
   - If yes, how can we avoid collisions?
8. **Is there a relation between table size and the number of items?**
9. **Why (key, value) pairs? Why not tuples?**
## Encoding of Information

### Problem

- How can keys of arbitrary objects be mapped to array indices that are whole numbers?
Problem

- How can keys of arbitrary objects be mapped to array indices that are whole numbers?
Problem

- How can an infinite number of keys be mapped to a finite number of indices?
Two stages of hash function

Problem

- How can an infinite number of keys be mapped to a finite number of indices?
Stage 1 of hash function: Hash code

- Consider bits as integer.
  \[
  \text{Hashcode}(\text{byte | short | char}) = 32\text{-bit int} \quad \triangleright \text{upsampling}
  \]
  \[
  \text{Hashcode}(\text{float}) = 32\text{-bit int} \quad \triangleright \text{change representation}
  \]
  \[
  \text{Hashcode}(\text{double}) = 32\text{-bit int} \quad \triangleright \text{downsampling}
  \]
  \[
  \text{Hashcode}(x_0, x_1, \ldots, x_{n-1}) = x_0 + x_1 + \cdots + x_{n-1} \quad \triangleright \text{sum}
  \]
  \[
  \text{Hashcode}(x_0, x_1, \ldots, x_{n-1}) = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1} \quad \triangleright \text{xor}
  \]

- Polynomial hash codes.
  \[
  \text{Hashcode}(x_0, x_1, \ldots, x_{n-1}) = x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1} \quad \triangleright \text{polynomial}
  \]

- Cyclic-shift hash codes.
  \[
  \text{Hashcode}_k(x) = \text{Rotate}(x, k \text{ bits}) \quad \triangleright \text{cyclic-shift}
  \]
  e.g.: \[
  \text{Hashcode}_2(111000) = 100011
  \]
A good compression function minimizes the number of collisions for a given set of distinct hash codes.

- **Division method.**
  \[
  \text{Compression}(i) = i \mod N
  \]
  > remainder
  
  $N \geq 1$ is the size of the bucket array.
  
  Often, $N$ being prime “spreads out” the distribution of primes.
  
  Ex. 1: Insert codes \{200, 205, \ldots, 600\} into $N$-sized array.
  Which is better: $N = 100$ or $N = 101$?
  
  Ex. 2: Insert multiple codes \{aN + b\} into $N$-sized array.
  Which is better: $N = \text{prime}$ or $N = \text{non-prime}$?

- **Multiply-Add-and-Divide (MAD) method.**
  \[
  \text{Compression}(i) = ((ai + b) \mod p) \mod N
  \]
  > remainder
  
  $N \geq 1$ is the size of the bucket array.
  
  $p$ is a prime number larger than $N$.
  
  $a, b$ are random integers from the range $[0, p - 1]$ with $a > 0$.
  
  Usually eliminates repeated patterns in the set of hash codes.
How do you implement the English dictionary search such that searching for a word takes $O(1)$ time on an average?
Hash table of English dictionary

### Key space

- \((\text{word, meaning})\)

### Hash space

<table>
<thead>
<tr>
<th>(\mathcal{H}(\text{word}))</th>
<th>(word, meaning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>499999</td>
<td></td>
</tr>
</tbody>
</table>
Collisions

\( (k_1, v_1) \)

\( k_1 \neq k_2 \)

\( (k_2, v_2) \)

\( \mathcal{H}(k_1) \)

\( \mathcal{H}(k_2) \)

Collisions

Entries

Hashes

\( 0 \)

\( 1 \)

\( \vdots \)

\( N - 2 \)

\( N - 1 \)
There are two major collision-handling schemes or collision-resolution strategies.

<table>
<thead>
<tr>
<th>Collision-handling scheme</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate chaining</td>
<td>Extra space (for secondary data structures)</td>
</tr>
<tr>
<td></td>
<td>Simpler implementation</td>
</tr>
<tr>
<td>Open addressing</td>
<td>No extra space</td>
</tr>
<tr>
<td></td>
<td>More complicated implementation</td>
</tr>
</tbody>
</table>
Separate chaining

- Have each bucket $A[j]$ store its own secondary container.
- We use **secondary data structures** (e.g. array list, linked list, balanced search trees, etc) for each bucket.
Separate chaining (via arraylist/linkedlist)

<table>
<thead>
<tr>
<th><strong>Put</strong>((key, value))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. hash ← Hash(key)</td>
</tr>
<tr>
<td>2. A[hash].AddLast((key, value)) ▶ A[hash] is a linked list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Get</strong>(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. hash ← Hash(key)</td>
</tr>
<tr>
<td>2. return A[hash].Get(key) ▶ returns value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Remove</strong>(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. hash ← Hash(key)</td>
</tr>
<tr>
<td>2. return A[hash].Remove(key) ▶ returns removed value</td>
</tr>
</tbody>
</table>
Open addressing

- All entries are stored in the bucket array itself.
- Strict requirement: Load factor must be at most 1.
- Useful in applications where there are space constraints, e.g.: smartphones and other small devices.
- Iteratively search the bucket \( A[(\text{HASH}(key) + f(i)) \mod N] \) for \( i = 0, 1, 2, 3, \ldots \) until finding an empty bucket.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear probing</td>
<td>( f(i) = i )</td>
</tr>
<tr>
<td>Quadratic probing</td>
<td>( f(i) = i^2 )</td>
</tr>
<tr>
<td>Double hashing</td>
<td>( f(i) = i \cdot \text{HASH2}(key) )</td>
</tr>
<tr>
<td></td>
<td>e.g. ( \text{HASH2}(key) = p - (key \mod p) ) for prime ( p &lt; N ).</td>
</tr>
<tr>
<td>Random generator</td>
<td>( f(i) = \text{RANDOM}(i, \text{HASH}(key)) )</td>
</tr>
</tbody>
</table>

Here, \( N \) should be a prime number.
Open addressing: Linear probing: Put

**Suppose** $\text{HASH}(key) = key \mod 10$

<table>
<thead>
<tr>
<th>Key</th>
<th>→</th>
<th>Hash</th>
<th>Put</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>→</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>→</td>
<td>1</td>
<td>41</td>
<td>18</td>
</tr>
<tr>
<td>22</td>
<td>→</td>
<td>2</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>32</td>
<td>→</td>
<td>2</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 probes)</td>
<td></td>
<td></td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>98</td>
<td>→</td>
<td>8</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 probes)</td>
<td></td>
<td></td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>58</td>
<td>→</td>
<td>8</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3 probes)</td>
<td></td>
<td></td>
<td>58</td>
<td>41</td>
</tr>
</tbody>
</table>

How many probes are required to insert 78?
Open addressing: Linear probing: Remove

- Suppose $\text{Hash}(key) = key \mod 10$

<table>
<thead>
<tr>
<th>Remove Key</th>
<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>58</td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>19</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
</tbody>
</table>

Hence, we cannot simply remove a found entry.

<table>
<thead>
<tr>
<th>Remove Key</th>
<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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>58</td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>19</td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>19</td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
<tr>
<td>19</td>
<td>58</td>
<td>41</td>
<td>22</td>
<td>32</td>
<td>78</td>
<td>19</td>
<td></td>
<td></td>
<td>18</td>
<td>98</td>
</tr>
</tbody>
</table>

Replace the deleted entry with the defunct object.
Open addressing: Linear probing

**Put**((key, value))

1. \( hash \leftarrow \text{Hash}(key); \ i \leftarrow 0 \)
2. \( \text{while} \ (hash + i) \mod N \neq \text{null} \ \text{and} \ i < N \ \text{do} \ i \leftarrow i + 1 \)
3. \( \text{if} \ i = N \ \text{then throw} \ \text{Bucket array is full} \)
4. \( \text{else} \ A[(hash + i) \mod N] \leftarrow (key, value) \)

**Get**(key)

1. \( hash \leftarrow \text{Hash}(key); \ i \leftarrow 0 \)
2. \( \text{while} \ (hash + i) \mod N \neq \text{null} \ \text{and} \ i < N \ \text{do} \)
3. \( \text{index} \leftarrow (hash + i) \mod N \)
4. \( \text{if} \ A[\text{index}].\text{key} = \text{key} \ \text{then return} \ A[\text{index}].\text{value} \)
5. \( i \leftarrow i + 1 \)
6. \( \text{return} \ \text{null} \)

**Remove**(key)

1. \( \text{index} \leftarrow \text{FindSlotForRemoval}(key) \)
2. \( \text{if} \ \text{index} < 0 \ \text{then return} \ \text{null} \)
3. \( \text{value} \leftarrow A[\text{index}].\text{value}; \ A[\text{index}] \leftarrow \text{defunct}; \ n \leftarrow n - 1 \)
4. \( \text{return} \ \text{value} \)
Complexity

- Suppose $N =$ bucket array size and $n =$ number of entries.
- Ratio $\lambda = n/N$ is called the load factor of the hash table.
- If $\lambda > 1$, rehash. Make sure $\lambda < 1$.
- Assuming good hash function, expected size of bucket is $O(\lceil \lambda \rceil)$.
- Separate chaining: Maintain $\lambda < 0.75$
  - Open addressing: Maintain $\lambda < 0.5$
- Assuming good hash function and $\lambda \in O(1)$, complexity of put, get, and remove is $O(1)$ expected time.
Symbol tables in compilers

1. double foo(int count)
2. {
3.     double sum = 0.0;
4.     for (int i = 1; i <= count; i++)
5.         sum += i;
6.     return sum;
7. }

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>function, double</td>
<td>global</td>
</tr>
<tr>
<td>count</td>
<td>int</td>
<td>function parameter</td>
</tr>
<tr>
<td>sum</td>
<td>double</td>
<td>block local</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>for-loop statement</td>
</tr>
</tbody>
</table>