Outline

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

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies
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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>dog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>cat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>pig</td>
<td></td>
<td></td>
<td></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**$(k)$: $A[k] \leftarrow$ NIL

Each operation is $\Theta(1)$.

Total space is $\Theta(M)$.

What sorting algorithm does this remind you of?
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** $h : U \to \{0, 1, \ldots, M - 1\}$.  
  (Commonly used: $h(k) = k \text{ mod } M$. 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

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

The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).
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:
  - multiple items at location (Chaining)
  - alternate slots in array (Open addressing)
    - many alternate slots (Probe sequence)
    - one alternate slot (Cuckoo Hashing)
  - Linear Probing
  - Double Hashing
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

\[ M = 11, \quad h(k) = k \mod 11 \]
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 $\frac{n}{M} := \alpha$.
  ($\alpha$ 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 $O(1 + \alpha)$. 
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 $\Theta(n)$. 
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.

Simplest method for open addressing: *linear probing*

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

for some hash function \( h \).
Linear probing example

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

```
0
1   45
2   13
3   92
4   49
5
6
7   7
8
9
10  43
```
Probe sequence operations

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

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

- 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 \( A = \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618 \).
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: $h(k) = 1 + [(M-1)(kA-[kA])]$

- **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

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

```cpp
cuckoo::insert(k, v)
1.   i ← 0
2.   do at most 2n times:
3.      if T_i[h_i(k)] is NIL
4.         T_i[h_i(k)] ← (k, v)
5.      return “success”
6.      swap((k, v), T_i[h_i(k)])
7.   i ← 1 − i
8.   return “failure to insert” // 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

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

\[
\begin{array}{c|c|c}
0 & 44 & 0 \\
1 & & 1 \\
2 & & 2 \\
3 & & 3 \\
4 & 59 & 4 \\
5 & & 5 \\
6 & & 6 \\
7 & & 7 \\
8 & & 8 \\
9 & 92 & 9 \\
10 & & 10 \\
\end{array}
\]
Cuckoo hashing discussions

- The two hash-tables need not be of the same size.
- Load factor $\alpha = \frac{n}{(\text{size of } T_0 + \text{size of } T_1)}$
- 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 $\alpha < 1/2$.

<table>
<thead>
<tr>
<th>Avg.-case costs:</th>
<th>search (unsuccessful)</th>
<th>insert</th>
<th>search (successful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Probing</td>
<td>$\frac{1}{(1 - \alpha)^2}$</td>
<td>$\frac{1}{(1 - \alpha)^2}$</td>
<td>$\frac{1}{1 - \alpha}$</td>
</tr>
<tr>
<td>Double Hashing</td>
<td>$\frac{1}{1 - \alpha}$</td>
<td>$\frac{1}{1 - \alpha}$</td>
<td>$\frac{1}{\alpha \log \left( \frac{1}{1 - \alpha} \right)}$</td>
</tr>
<tr>
<td>Cuckoo Hashing</td>
<td>$\frac{1}{(\text{worst-case})}$</td>
<td>$\frac{\alpha}{(1 - 2\alpha)^2}$</td>
<td>$\frac{1}{(\text{worst-case})}$</td>
</tr>
</tbody>
</table>

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 $M \cdot n$ keys, then there are $n$ keys that all hash to the same value.
- 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”.

Biedl, Schost, Veksler (SCS, UW)
Multi-dimensional Data

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

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

\[
A \cdot P \cdot P \cdot L \cdot E \rightarrow (65, 80, 80, 76, 69) \quad \text{(ASCII)}
\]
\[
\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 68R^0
\]
\[
\text{(for some radix } R, \text{ e.g. } R = 255)
\]

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

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

\[
\left(\left(\left(\left(\left(\left((65R+80) \mod M\right)R+80\right) \mod M\right)R+76\right) \mod M\right)R+69\right) \mod M
\]
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