Module 7: Hashing

CS 240 - Data Structures and Data Management

Sajed Haque    Veronika Irvine    Taylor Smith
Based on lecture notes by many previous cs240 instructors

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

Spring 2017
Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ items. Is this the best possible?

Theorem

In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.

Proof:

Similar to lower bound for sorting. Any algorithm defines a binary decision tree with comparisons at the nodes and actions at the leaves. There are at least $n + 1$ different actions (return an item, or "not found"). So there are $\Omega(n)$ leaves, and therefore the height is $\Omega(\log n)$. □
Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ items. Is this the best possible?

**Theorem:** In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.
Lower bound for search

The fastest implementations of the dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing $n$ items. Is this the best possible?

**Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.

**Proof**: Similar to lower bound for sorting.

Any algorithm defines a binary decision tree with comparisons at the nodes and actions at the leaves.

There are at least $n+1$ different actions (return an item, or “not found”). So there are $\Omega(n)$ leaves, and therefore the height is $\Omega(\log n)$. □
Direct Addressing

**Requirement**: For a given $M \in \mathbb{N}$, every key $k$ is an integer with $0 \leq k < M$.

**Data structure**: An array of *values* $A$ with size $M$

- $search(k)$: Check whether $A[k]$ is empty
- $insert(k, v)$: $A[k] \leftarrow v$
- $delete(k)$: $A[k] \leftarrow empty$

Each operation is $\Theta(1)$. Total storage is $\Theta(M)$. What sorting algorithm does this remind you of?

Counting Sort
Direct Addressing

**Requirement**: For a given $M \in \mathbb{N}$, every key $k$ is an integer with $0 \leq k < M$.

**Data structure**: An array of values $A$ with size $M$

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

*Each operation is $\Theta(1)$.*

Total storage is $\Theta(M)$.

What sorting algorithm does this remind you of?
Direct Addressing

**Requirement**: For a given $M \in \mathbb{N}$, every key $k$ is an integer with $0 \leq k < M$.

**Data structure**: An array of values $A$ with size $M$

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

*Each operation is $\Theta(1)$.*  
Total storage is $\Theta(M)$.

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

Direct addressing isn’t possible if keys are not integers. And the storage is very wasteful if $n \ll M$.

Say keys come from some universe $U$. Use a hash function $h : U \rightarrow \{0, 1, \ldots, M - 1\}$. Generally, $h$ is not injective, so many keys can map to the same integer.
Hashing

Direct addressing isn’t possible if keys are not integers. And the storage is very wasteful if $n \ll M$.

Say keys come from some universe $U$. Use a hash function $h : U \to \{0, 1, \ldots, M - 1\}$. Generally, $h$ is not injective, so many keys can map to the same integer.

**Hash table Dictionary**: Array $T$ of size $M$ (the hash table). An item with key $k$ is stored in $T[h(k)]$. *search, insert, and delete* should all cost $O(1)$.

Challenges:
- Choosing a good hash function (later)
- Dealing with *collisions* (when $h(k_1) = h(k_2)$)
Collision Resolution

Even the best hash function may have collisions: when we want to insert \((k, v)\) into the table, but \(T[h(k)]\) is already occupied.

Two basic strategies:
- Allow multiple items at each table location (buckets)
- Allow each item to go into multiple locations (open addressing)

We will examine the average cost of search, insert, delete, in terms of \(n, M\), and/or the load factor \(\alpha = n/M\).

We probably want to rebuild the whole hash table and change the value of \(M\) when the load factor gets too large or too small. This is called rehashing, and should cost roughly \(\Theta(M + n)\).
Chaining

Each table entry is a *bucket* containing 0 or more KVPs. This could be implemented by any dictionary (even another hash table!).

The simplest approach is to use an unsorted linked list in each bucket. This is called collision resolution by *chaining*.

- **search**($k$): Look for key $k$ in the list at $T[h(k)]$.
- **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

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

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

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

- **insert\((41)\)**

<table>
<thead>
<tr>
<th>Insertion Point</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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><strong>Red</strong></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>

\[ h(41) = 8 \]
Chaining example

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

*insert*(41)

\[ h(41) = 8 \]
Chaining example

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

\[ \text{insert}(46) \]

\( h(46) = 2 \)

\[
\begin{array}{c}
0 \\
1 & 45 \\
2 & 13 \\
3 \\
4 & 92 \\
5 & 49 \\
6 \\
7 & 7 \\
8 & 41 \\
9 \\
10 & 43 \\
\end{array}
\]
Chaining example

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

**insert(46)**

\[ h(46) = 2 \]

Inserting 46 into the hash table.
Chaining example

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

**insert(16)**

\[ h(16) = 5 \]
**Chaining example**

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

*insert*(79)

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

Recall the load balance $\alpha = n/M$.

Assuming uniform hashing, average bucket size is exactly $\alpha$.

Analysis of operations:

- **search**: $\Theta(1 + \alpha)$ average-case, $\Theta(n)$ worst-case
- **insert**: $O(1)$ worst-case, since we always insert in front.
- **delete**: Same cost as **search**: $\Theta(1 + \alpha)$ average, $\Theta(n)$ worst-case

If we maintain $M \in \Theta(n)$, then average costs are all $O(1)$.
This is typically accomplished by rehashing whenever $n < c_1 M$ or $n > c_2 M$, for some constants $c_1, c_2$ with $0 < c_1 < c_2$. 
Open addressing

**Main idea:** Each hash table entry holds only one item, but any key $k$ can go in multiple locations.

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

*delete* becomes problematic; we must distinguish between *empty* and *deleted* locations.
Open addressing

**Main idea:** Each hash table entry holds only one item, but any key $k$ can go in multiple locations.

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

*delete* becomes problematic; we must distinguish between *empty* and *deleted* locations.

Simplest idea: *linear probing*

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

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear probing example

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

**insert\((41)\)**

- \( h(41, 0) = 8 \)

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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>
Linear probing example

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

**insert**(84)

\[ h(84, 0) = 7 \]

```
  0   45
  1
  2   13
  3
  4   92
  5   49
  6
  7   7
  8   41
  9
 10   43
```
Linear probing example

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

**insert(84)**

\[ h(84, 1) = 8 \]
Linear probing example

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

*insert*(84)

\[ h(84, 2) = 9 \]

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>
<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>
<td></td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></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>
<td>92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></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>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Haque, Irvine, Smith (SCS, UW)  
CS240 - Module 7  
Spring 2017
Linear probing example

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

**insert(20)**

\[ h(20, 0) = 9 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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>84</td>
</tr>
<tr>
<td>10</td>
<td>43</td>
</tr>
</tbody>
</table>
Linear probing example

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

\textit{insert}(20)

\[ h(20, 1) = 10 \]
Linear probing example

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

\textit{insert}(20)

\[ h(20, 2) = 0 \]
Linear probing example

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

**delete(43)**

\[ h(43, 0) = 10 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td><strong>deleted</strong></td>
</tr>
</tbody>
</table>
Linear probing example

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

\textbf{search}(63)

\[ h(63, 0) = 8 \]

\begin{align*}
0 & \quad 20 \\
1 & \quad 45 \\
2 & \quad 13 \\
3 & \\
4 & \quad 92 \\
5 & \quad 49 \\
6 & \\
7 & \quad 7 \\
8 & \quad 41 \\
9 & \quad 84 \\
10 & \quad \text{deleted}
\end{align*}
Linear probing example

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

**search(63)**

\[ h(63, 1) = 9 \]

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>deleted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear probing example

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

**search(63)**

\[ h(63, 2) = 10 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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>84</td>
</tr>
<tr>
<td>10</td>
<td><strong>deleted</strong></td>
</tr>
</tbody>
</table>
Linear probing example

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

\textit{search}(63)

\[ h(63, 3) = 0 \]

```
0  20
1  45
2  13
3
4  92
5  49
6
7  7
8  41
9  84
10 deleted
```
# Linear probing example

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

**search(63)**

\[ h(63, 4) = 1 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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>84</td>
</tr>
<tr>
<td>10</td>
<td><strong>deleted</strong></td>
</tr>
</tbody>
</table>
Linear probing example

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

\[ search(63) \]

\[ h(63, 5) = 2 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td></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>84</td>
</tr>
<tr>
<td>10</td>
<td>deleted</td>
</tr>
</tbody>
</table>
Linear probing example

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

**search(63)**

\[ h(63, 6) = 3 \]

<table>
<thead>
<tr>
<th>(k)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>92</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td></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>84</td>
</tr>
<tr>
<td>10</td>
<td><strong>deleted</strong></td>
</tr>
</tbody>
</table>
Double Hashing

Say we have two hash functions $h_1, h_2$ that are independent.

So, under uniform hashing, we assume the probability that a key $k$ has $h_1(k) = a$ and $h_2(k) = b$, for any particular $a$ and $b$, is

$$\frac{1}{M^2}.$$

For double hashing, define $h(k, i) = h_1(k) + i \cdot h_2(k) \mod M$.

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

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor k/2 \rfloor \mod 11 \]

\[
\begin{array}{c}
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45</td>
<td>13</td>
<td></td>
<td>92</td>
<td>49</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>43</td>
</tr>
</tbody>
</table>
\end{array}
\]
Double hashing example

\( M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor k/2 \rfloor \mod 11 \)

insert(41)

\( h_1(41) = 8 \)

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

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor k/2 \rfloor \mod 11 \]

**insert** (84)

\[ h_1(84) = 7 \]
\[ h(84, 0) = 7 \]
Double hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor k/2 \rfloor \mod 11 \]

**insert(84)**

\[ h_1(84) = 7 \]
\[ h(84, 0) = 7 \]
\[ h_2(84) = 9 \]
\[ h(84, 1) = 5 \]

```
  0  
  1   45  
  2   13  
  3  
  4   92  
  5   49  
  6  
  7   7  
  8   41  
  9  
 10  43  
```
Double hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor k/2 \rfloor \mod 11 \]

\textit{insert}(84)

\[
\begin{align*}
h_1(84) &= 7 \\
h(84, 0) &= 7 \\
h_2(84) &= 9 \\
h(84, 1) &= 5 \\
h(84, 2) &= 3
\end{align*}
\]
Cuckoo hashing

This is a relatively new idea from Pagh and Rodler in 2001.

Again, we use two independent hash functions $h_1, h_2$. The idea is to *always* insert a new item into $h_1(k)$. This might “kick out” another item, which we then attempt to re-insert into its alternate position.

Insertion might not be possible if there is a loop. In this case, we have to rehash with a larger $M$.

The big advantage is that an element with key $k$ can only be in $T[h_1(k)]$ or $T[h_2(k)]$. 
Cuckoo hashing insertion

\[
cuckoo-insert(T, x)
\]

\(T\): hash table, \(x\): new item to insert

1. \(y \leftarrow x, \quad i \leftarrow h_1(x.key)\)
2. \textbf{do} at most \(n\) times:
3. \hspace{1em} \textbf{swap}(y, T[i])
4. \hspace{1em} \textbf{if} \(y\) is “empty” \textbf{then return} “success”
5. \hspace{1em} \textbf{if} \(i = h_1(y.key)\) \textbf{then} \(i \leftarrow h_2(y.key)\)
6. \hspace{1em} \textbf{else} \(i \leftarrow h_1(y.key)\)
7. \hspace{1em} \textbf{return} “failure”
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

insert(51)

\[ y.key = 51 \]
\[ i = 7 \]

\[ h_1(y.key) = 7 \]
\[ h_2(y.key) = 5 \]
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

\textit{insert}(51)

\begin{align*}
y.key &= \text{ } \\
i &= \text{ } \\
h_1(y.key) &= \text{ } \\
h_2(y.key) &= \text{ }
\end{align*}

\begin{array}{c|c}
0 & 44 \\
1 & \\
2 & \\
3 & \\
4 & 26 \\
5 & \\
6 & \\
7 & 51 \\
8 & \\
9 & 92 \\
10 & \\
\end{array}
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

*insert*(95)

\( y.key = 95 \)

\( i = 7 \)

\( h_1(y.key) = 7 \)

\( h_2(y.key) = 7 \)
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert**(95)

\( y.key = 51 \)
\( i = 5 \)

\( h_1(y.key) = 7 \)
\( h_2(y.key) = 5 \)
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

**insert** (95)

\[
\begin{align*}
    y.\text{key} &= \quad i &= \\
    h_1(y.\text{key}) &= \quad 44 \\
    h_2(y.\text{key}) &= \quad 51
\end{align*}
\]


Cuckoo hashing example

\( M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \)

\textit{insert}(97)

\( y.key = 97 \)

\( i = 9 \)

\( h_1(y.key) = 9 \)

\( h_2(y.key) = 10 \)

<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>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

Insert(97)

y.key = 92

\[ i = 4 \]

\[ h_1(y.key) = 4 \]

\[ h_2(y.key) = 9 \]
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

\textit{insert}(97)

\( y.key = 26 \)

\( i = 0 \)

\( h_1(y.key) = 4 \)

\( h_2(y.key) = 0 \)
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor \]

**insert**(97)

\[ y.key = 44, \quad i = 2 \]

\[ h_1(y.key) = 0, \quad h_2(y.key) = 2 \]
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \mod 11, \quad h_2(k) = \left\lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \right\rfloor \]

**insert(97)**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo hashing example

\[ M = 11, \quad h_1(k) = k \ mod \ 11, \quad h_2(k) = \lfloor 11(\phi k - \lfloor \phi k \rfloor) \rfloor \]

search(26)

\[ h_1(26) = 4 \]
\[ h_2(26) = 0 \]
Choosing a good hash function

**Uniform Hashing Assumption**: Each hash function value is equally likely.

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

We can get good performance by following a few rules.

A good hash function should:

- be very efficient to compute
- be unrelated to any possible patterns in the data
- depend on all parts of the key
Basic hash functions

If all keys are integers (or can be mapped to integers), the following two approaches tend to work well:

**Division method**: \( h(k) = k \mod M \).
We should choose \( M \) to be a prime.

**Multiplication method**: \( h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor \),
for some constant floating-point number \( A \) with \( 0 < A < 1 \).
Knuth suggests \( A = \varphi = \frac{\sqrt{5} - 1}{2} \approx 0.618 \).
Multi-dimensional Data

What if the keys are multi-dimensional, such as strings?

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

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

We combine this with a standard hash function $h : \mathbb{N} \rightarrow \{0, 1, 2, \ldots, M - 1\}$.

With $h(f(k))$ as the hash values, we then use any standard hash table.

**Note:** computing each $h(f(k))$ takes $\Omega($length of $w$) time.
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
- No wasted space
- Never need to rebuild the entire structure

Advantages of Hash Tables

- $O(1)$ cost, but only on average
- Flexible load factor parameters
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete