7.7 Hashing

Dictionary:
- \textbf{S. insert}(x): Insert an element \( x \).
- \textbf{S. delete}(x): Delete the element pointed to by \( x \).
- \textbf{S. search}(k): Return a pointer to an element \( e \) with \( \text{key}[e] = k \) in \( S \) if it exists; otherwise return \textit{null}.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object \( x \) with key \( k \) is determined by successively comparing \( k \) to split-elements.

\textbf{Hashing} tries to \textit{directly} compute the memory location from the given key. The goal is to have constant search time.

---

7.7 Hashing

Definitions:
- Universe \( U \) of keys, e.g., \( U \subseteq \mathbb{N}_0 \). \( U \) very large.
- Set \( S \subseteq U \) of keys, \( |S| = m \leq |U| \).
- Array \( T[0, \ldots, n - 1] \) hash-table.
- Hash function \( h : U \rightarrow [0, \ldots, n - 1] \).

The hash-function \( h \) should fulfill:
- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

---

7.7 Hashing

Direct Addressing

Ideally the hash function maps all keys to different memory locations.

This special case is known as Direct Addressing. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

---

7.7 Hashing

Perfect Hashing

Suppose that we \textit{know} the set \( S \) of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.

Such a hash function \( h \) is called a \textbf{perfect hash function} for set \( S \).
Collisions

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

**Problem: Collisions**

Usually the universe $U$ is much larger than the table-size $n$.

Hence, there may be two elements $k_1, k_2$ from the set $S$ that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

**Lemma 1**

The probability of having a collision when hashing $m$ elements into a table of size $n$ under uniform hashing is at least

$$1 - e^{-m(m-1)/2n} \approx 1 - e^{-m^2/2n}.$$

**Uniform hashing:**

Choose a hash function uniformly at random from all functions $f : U \rightarrow [0, \ldots, n-1]$.

**Proof.**

Let $A_{m,n}$ denote the event that inserting $m$ keys into a table of size $n$ does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}}.$$

Here the first equality follows since the $\ell$-th element that is hashed has a probability of $\frac{n-\ell+1}{n}$ to not generate a collision under the condition that the previous elements did not induce collisions.

The inequality $1-x \leq e^{-x}$ is derived by stopping the Taylor-expansion of $e^{-x}$ after the second term.
Resolving Collisions

The methods for dealing with collisions can be classified into the two main types

- open addressing, aka. closed hashing
- hashing with chaining, aka. closed addressing, open hashing.

There are applications e.g. computer chess where you do not resolve collisions at all.

Hashing with Chaining

Let $A$ denote a strategy for resolving collisions. We use the following notation:

- $A^+$ denotes the average time for a **successful** search when using $A$;
- $A^-$ denotes the average time for an **unsuccessful** search when using $A$;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

The time required for an unsuccessful search is $1$ plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if $A$ is the collision resolving strategy “Hashing with Chaining” we have

$$A^- = 1 + \alpha.$$
Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key \( k \) in the hash-table and ask for the search-time for \( k \).

This is 1 plus the number of elements that lie before \( k \) in \( k \)'s list.

Let \( k_\ell \) denote the \( \ell \)-th key inserted into the table.

Let for two keys \( k_i \) and \( k_j \), \( X_{ij} \) denote the indicator variable for the event that \( k_i \) and \( k_j \) hash to the same position. Clearly, \( \Pr[X_{ij} = 1] = 1/n \) for uniform hashing.

The expected successful search cost is

\[
E\left[ \frac{1}{m} \sum_{i=1}^{m} \left( 1 + \sum_{j=i+1}^{m} X_{ij} \right) \right] = \frac{1}{m} \sum_{i=1}^{m} \left( 1 + \sum_{j=i+1}^{m} \frac{1}{n} \right)
\]

\[
= 1 + \frac{1}{mn} \sum_{i=1}^{m} (m - i)
\]

\[
= 1 + \frac{1}{mn} \left( m^2 - \frac{m(m + 1)}{2} \right)
\]

\[
= 1 + \frac{m - 1}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2m}.
\]

Hence, the expected cost for a successful search is \( A^+ \leq 1 + \frac{\alpha}{2} \).

Open Addressing

All objects are stored in the table itself.

Define a function \( h(k, j) \) that determines the table-position to be examined in the \( j \)-th step. The values \( h(k, 0), \ldots, h(k, n-1) \) must form a permutation of \( 0, \ldots, n-1 \).

\textbf{Search}(k): Try position \( h(k, 0) \); if it is empty your search fails; otw. continue with \( h(k, 1), h(k, 2), \ldots \).

\textbf{Insert}(x): Search until you find an empty slot; insert your element there. If your search reaches \( h(k, n-1) \), and this slot is non-empty then your table is full.

Hashing with Chaining

Disadvantages:

- pointers increase memory requirements
- pointers may lead to bad cache efficiency

Advantages:

- no à priori limit on the number of elements
- deletion can be implemented efficiently
- by using balanced trees instead of linked list one can also obtain worst-case guarantees.
Open Addressing

Choices for \( h(k,j) \):

- **Linear probing:**
  \[ h(k,i) = h(k) + i \mod n \]
  (sometimes: \( h(k,i) = h(k) + c_i \mod n \)).

- **Quadratic probing:**
  \[ h(k,i) = h(k) + c_1i + c_2i^2 \mod n. \]

- **Double hashing:**
  \[ h(k,i) = h_1(k) + ih_2(k) \mod n. \]

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing \( h_2(k) \) must be relatively prime to \( n \) (teilerfremd); for quadratic probing \( c_1 \) and \( c_2 \) have to be chosen carefully).

### Linear Probing

- **Advantage:** **Cache-efficiency.** The new probe position is very likely to be in the cache.
- **Disadvantage:** **Primary clustering.** Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

**Lemma 2**

Let \( l \) be the method of linear probing for resolving collisions:

\[
L^+ \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right)
\]

\[
L^- \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2}\right)
\]

### Double Hashing

- **Any probe into the hash-table usually creates a cache-miss.**

**Lemma 4**

Let \( A \) be the method of double hashing for resolving collisions:

\[
D^+ \approx \frac{1}{\alpha} \ln \left(\frac{1}{1 - \alpha}\right)
\]

\[
D^- \approx \frac{1}{1 - \alpha}
\]
Open Addressing

Some values:

<table>
<thead>
<tr>
<th>α</th>
<th>Linear Probing</th>
<th>Quadratic Probing</th>
<th>Double Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L⁺</td>
<td>L⁻</td>
<td>Q⁺</td>
</tr>
<tr>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
<td>1.44</td>
</tr>
<tr>
<td>0.9</td>
<td>5.5</td>
<td>50.5</td>
<td>2.85</td>
</tr>
<tr>
<td>0.95</td>
<td>10.5</td>
<td>200.5</td>
<td>3.52</td>
</tr>
</tbody>
</table>

Analysis of Idealized Open Address Hashing

We analyze the time for a search in a very idealized Open Addressing scheme.

- The probe sequence \( h(k,0), h(k,1), h(k,2), \ldots \) is equally likely to be any permutation of \( \langle 0,1,\ldots,n-1 \rangle \).

Analysis of Idealized Open Address Hashing

Let \( X \) denote a random variable describing the number of probes in an unsuccessful search.

Let \( A_i \) denote the event that the \( i \)-th probe occurs and is to a non-empty slot.

\[
\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}] = \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]
\]

\[
\Pr[X \geq i] = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2}
\]

\[
\leq \left( \frac{m}{n} \right)^{i-1} = \alpha^{i-1} .
\]
Analysis of Idealized Open Address Hashing

\[
E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] = \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}.
\]

\[
\frac{1}{1 - \alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots
\]

The \(j\)-th rectangle appears in both sums \(j\) times. (\(j\) times in the first due to multiplication with \(j\); and \(j\) times in the second for summands \(i = 1, 2, \ldots, j\)).

---

Analysis of Idealized Open Address Hashing

\[
i = 3
\]

\[
\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]
\]

The \(j\)-th rectangle appears in both sums \(j\) times. (\(j\) times in the first due to multiplication with \(j\); and \(j\) times in the second for summands \(i = 1, 2, \ldots, j\)).

---

Analysis of Idealized Open Address Hashing

\[
i = 4
\]

\[
\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]
\]

The \(j\)-th rectangle appears in both sums \(j\) times. (\(j\) times in the first due to multiplication with \(j\); and \(j\) times in the second for summands \(i = 1, 2, \ldots, j\)).

---

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for \(k\) is equal to the number of probes made in an unsuccessful search for \(k\) at the time that \(k\) is inserted.

Let \(k\) be the \(i + 1\)-st element. The expected time for a search for \(k\) is at most

\[
\frac{1}{n-i} = \frac{n}{n-i}.
\]

\[
\frac{1}{m} \sum_{i=0}^{m-1} \frac{n}{n-i} = \frac{n}{m} \sum_{i=0}^{m-1} \frac{1}{n-i} = \frac{1}{\alpha} \sum_{k=n}^{n+m+1} \frac{1}{k}
\]

\[
\leq \frac{1}{\alpha} \int_{n-m}^{n} \frac{1}{x} \, dx = \frac{1}{\alpha} \ln \frac{n}{n-m} = \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}.
\]
Analysis of Idealized Open Address Hashing

Deletions in HashTables

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

Deletions in HashTables

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions for Linear Probing

- For Linear Probing one can delete elements without using deletion-markers.
- Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.
Deletions for Linear Probing

**Algorithm 12 delete** \((p)\)

1: \(T[p] \leftarrow \text{null}\)
2: \(p \leftarrow \text{succ}(p)\)
3: while \(T[p] \neq \text{null}\) do
4: \(y \leftarrow T[p]\)
5: \(T[p] \leftarrow \text{null}\)
6: \(p \leftarrow \text{succ}(p)\)
7: \(\text{insert}(y)\)

\(p\) is the index into the table-cell that contains the object to be deleted.

Pointers into the hash-table become invalid.

---

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that \(h\) is chosen randomly from all functions \(f : U \rightarrow \{0, \ldots, n - 1\}\) is clearly unrealistic as there are \(n^{|U|}\) such functions. Even writing down such a function would take \(|U| \log n\) bits.

Universal hashing tries to define a set \(\mathcal{H}\) of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from \(\mathcal{H}\).

---

**Definition 5**

A class \(\mathcal{H}\) of hash-functions from the universe \(U\) into the set \(\{0, \ldots, n - 1\}\) is called universal if for all \(u_1, u_2 \in U\) with \(u_1 \neq u_2\)

\[
\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n},
\]

where the probability is w. r. t. the choice of a random hash-function from set \(\mathcal{H}\).

Note that this means that the probability of a collision between two arbitrary elements is at most \(\frac{1}{n}\).

---

**Definition 6**

A class \(\mathcal{H}\) of hash-functions from the universe \(U\) into the set \(\{0, \ldots, n - 1\}\) is called 2-independent (pairwise independent) if the following two conditions hold

- For any key \(u \in U\), and \(t \in \{0, \ldots, n - 1\}\) \(\Pr[h(u) = t] = \frac{1}{n}\), i.e., a key is distributed uniformly within the hash-table.
- For all \(u_1, u_2 \in U\) with \(u_1 \neq u_2\), and for any two hash-positions \(t_1, t_2\):

\[
\Pr[h(u_1) = t_1 \land h(u_2) = t_2] \leq \frac{1}{n^2}.
\]

This requirement clearly implies a universal hash-function.
Definition 7
A class $\mathcal{H}$ of hash-functions from the universe $U$ into the set $\{0, \ldots, n-1\}$ is called $k$-independent if for any choice of $\ell \leq k$ distinct keys $u_1, \ldots, u_\ell \in U$, and for any set of $\ell$ not necessarily distinct hash-positions $t_1, \ldots, t_\ell$:

$$\Pr[h(u_1) = t_1 \land \cdots \land h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell},$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Lemma 9
The class $\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$ is a universal class of hash-functions from $U$ to $\{0, \ldots, n-1\}$.

Proof.
Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

1. $ax + b \neq ay + b \pmod{p}$

If $x \neq y$ then $(x - y) \neq 0 \pmod{p}$.

Multiplying with $a \neq 0 \pmod{p}$ gives $a(x - y) \neq 0 \pmod{p}$.

where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).
Universal Hashing

The hash-function does not generate collisions before the \( (\mod n) \) operation. Furthermore, every choice \((a, b)\) is mapped to a different pair \((t_x, t_y)\) with \( t_x := ax + b \) and \( t_y := ay + b \).

This holds because we can compute \( a \) and \( b \) when given \( t_x \) and \( t_y \):

\[
\begin{align*}
t_x &= ax + b \\
(t_x - t_y) &= a(x - y) \\
t_y &= ay + b \\
a &= (t_x - t_y)(x - y)^{-1} \\
b &= t_y - ay
\end{align*}
\]

\( \mod p \)

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs \((a, b), a \neq 0\)) and pairs \((t_x, t_y), t_x \neq t_y\).

Therefore, we can view the first step (before the \( \mod n \) operation) as choosing a pair \((t_x, t_y), t_x \neq t_y\) uniformly at random.

What happens when we do the \( \mod n \) operation?

Fix a value \( t_x \). There are \( p - 1 \) possible values for choosing \( t_y \).

From the range \( 0, \ldots, p - 1 \) the values \( t_x, t_x + n, t_x + 2n, \ldots \) map to \( t_x \) after the modulo-operation. These are at most \( \lceil p/n \rceil \) values.

As \( t_y \neq t_x \) there are

\[
\left\lfloor \frac{p}{n} \right\rfloor - 1 \leq \frac{p}{n} + \frac{n - 1}{n} - 1 = \frac{p - 1}{n}
\]

possibilities for choosing \( t_y \) such that the final hash-value creates a collision.

This happens with probability at most \( \frac{1}{n} \).

It is also possible to show that \( \mathcal{H} \) is an (almost) pairwise independent class of hash-functions.

\[
\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p - 1)} \leq \Pr_{t_x=t_y \in \mathbb{Z}_p} \left[ t_x \mod n = h_1 \land t_y \mod n = h_2 \right] \leq \frac{\left\lceil \frac{p}{n} \right\rceil^2}{p(p - 1)}
\]

Note that the middle is the probability that \( h(x) = h_1 \) and \( h(y) = h_2 \). The total number of choices for \((t_x, t_y)\) is \( p(p - 1) \).

The number of choices for \( t_x \) \((t_y)\) such that \( t_x \mod n = h_1 \) \((t_y \mod n = h_2)\) lies between \( \left\lfloor \frac{p}{n} \right\rfloor \) and \( \left\lceil \frac{p}{n} \right\rceil \).
**Definition 10**
Let \( d \in \mathbb{N}; q \geq (d + 1)n \) be a prime; and let \( \bar{a} \in \{0, \ldots, q - 1\}^{d+1} \). Define for \( x \in \{0, \ldots, q - 1\} \)
\[
h_{\bar{a}}(x) := \left( \sum_{i=0}^{d} a_i x^i \mod q \right) \mod n.
\]
Let \( \mathcal{H}_n^d := \{ h_{\bar{a}} \mid \bar{a} \in \{0, \ldots, q - 1\}^{d+1} \} \). The class \( \mathcal{H}_n^d \) is \((e, d + 1)\)-independent.

Note that in the previous case we had \( d = 1 \) and chose \( a_d \neq 0 \).

---

For the coefficients \( \bar{a} \in \{0, \ldots, q - 1\}^{d+1} \) let \( f_{\bar{a}} \) denote the polynomial
\[
f_{\bar{a}}(x) = \left( \sum_{i=0}^{d} a_i x^i \right) \mod q
\]
The polynomial is defined by \( d + 1 \) distinct points.

Now, we choose \( d - \ell + 1 \) other inputs and choose their value arbitrarily. We have \( q^{d-\ell+1} \) possibilities to do this.

Therefore we have
\[
|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \left( \frac{q}{n} \right)^\ell \cdot q^{d-\ell+1}
\]
possibilities to choose \( \bar{a} \) such that \( h_{\bar{a}} \in A_\ell \).
Universal Hashing

Therefore the probability of choosing $h_a$ from $A_\ell$ is only

$$\frac{\frac{q}{n}^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \leq \frac{(\frac{q+n}{n})^\ell}{q^\ell} \cdot \frac{1}{n^\ell} \leq \frac{1}{n^\ell} \leq \frac{e}{n^\ell}.$$  

This shows that the $\mathcal{H}$ is $(e,d+1)$-universal.

The last step followed from $q \geq (d+1)n$, and $\ell \leq d+1$.

Perfect Hashing

Suppose that we know the set $S$ of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.

Let $m = |S|$. We could simply choose the hash-table size very large so that we don’t get any collisions.

Using a universal hash-function the expected number of collisions is

$$E[\# \text{Collisions}] = \left( \frac{m}{2} \right) \cdot \frac{1}{n}.$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.
Perfect Hashing

The total memory that is required by all hash-tables is \( O(\sum_j m_j^2) \). Note that \( m_j \) is a random variable.

\[
E\left[ \sum_j m_j^2 \right] = E\left[ 2 \sum_j \left( \frac{m_j}{2} \right)^2 + \sum_j m_j \right]
= 2 E\left[ \sum_j \left( \frac{m_j}{2} \right)^2 \right] + E\left[ \sum_j m_j \right]
\]

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

\[
= 2 \left( \frac{m}{2} \right) \frac{1}{m} + m = 2m - 1.
\]

Perfect Hashing

We need only \( O(m) \) time to construct a hash-function \( h \) with \( \sum_j m_j^2 = O(4m) \), because with probability at least \( 1/2 \) a random function from a universal family will have this property.

Then we construct a hash-table \( h_j \) for every bucket. This takes expected time \( O(m_j) \) for every bucket. A random function \( h_j \) is collision-free with probability at least \( 1/2 \). We need \( O(m_j) \) to test this.

We only need that the hash-functions are chosen from a universal family!!!
Cuckoo Hashing

Insert:

\[
\begin{array}{c}
\emptyset \\
\emptyset \\
x_1 \\
\emptyset \\
x_4 \\
x_6 \\
\emptyset \\
x_7 \\
\emptyset \\
x_9 \\
\emptyset \\
x_3 \\
\emptyset \\
x_7 \\
\emptyset \\
x_6 \\
x_1
\end{array}
\]

\[
\begin{array}{c}
\emptyset \\
\emptyset \\
x_2 \\
\emptyset \\
x_5 \\
x_8 \\
\emptyset \\
x_7 \\
\emptyset \\
x_7 \\
\emptyset \\
x_3 \\
\emptyset \\
x_7 \\
\emptyset \\
x_6 \\
x_1
\end{array}
\]

Algorithm 13 Cuckoo-Insert(x)
1: if \( T_1[h_1(x)] = x \lor T_2[h_2(x)] = x \) then return
2: steps ← 1
3: while steps ≤ maxsteps do
4: exchange x and \( T_1[h_1(x)] \)
5: if x = null then return
6: exchange x and \( T_2[h_2(x)] \)
7: if x = null then return
8: steps ← steps + 1
9: rehash() // change hash-functions; rehash everything
10: Cuckoo-Insert(x)

Cuckoo Hashing

- We call one iteration through the while-loop a step of the algorithm.
- We call a sequence of iterations through the while-loop without the termination condition becoming true a phase of the algorithm.
- We say a phase is successful if it is not terminated by the maxstep-condition, but the while loop is left because \( x = \text{null} \).

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after \( \text{maxsteps} \) steps).

Formally what is the probability to enter an infinite loop that touches \( s \) different keys?
Cuckoo Hashing: Insert

\[ x = x_1 \]

\[ x = x_2 \]

\[ x = x_3 \]

\[ x = x_4 \]

\[ x = x_5 \]

\[ x = x_6 \]

\[ x = x_7 \]

\[ x = x_8 \]

A cycle-structure of size \( s \) is defined by

\( \triangleright s - 1 \) different cells (alternating btw. cells from \( T_1 \) and \( T_2 \)).

\( \triangleright s \) distinct keys \( x = x_1, x_2, \ldots, x_s \), linking the cells.

\( \triangleright \) The leftmost cell is “linked forward” to some cell on the right.

\( \triangleright \) The rightmost cell is “linked backward” to a cell on the left.

\( \triangleright \) One link represents key \( x \); this is where the counting starts.

Cuckoo Hashing

A cycle-structure is active if for every key \( x_\ell \) (linking a cell \( p_i \) from \( T_1 \) and a cell \( p_j \) from \( T_2 \)) we have

\[ h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j \]

Observation:
If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size \( s \geq 3 \).

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size \( s \) correctly map into their \( T_1 \)-cell?

This probability is at most \( \frac{\mu}{n^s} \) since \( h_1 \) is a \((\mu, s)\)-independent hash-function.

What is the probability that all keys in the cycle-structure of size \( s \) correctly map into their \( T_2 \)-cell?

This probability is at most \( \frac{\mu}{n^s} \) since \( h_2 \) is a \((\mu, s)\)-independent hash-function.

These events are independent.
Cuckoo Hashing

The probability that a given cycle-structure of size $s$ is active is at most $\frac{\mu^2}{m^2}$.

What is the probability that there exists an active cycle structure of size $s$?

The number of cycle-structures of size $s$ is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1}.$$ 

- There are at most $s^2$ possibilities where to attach the forward and backward links.
- There are at most $s$ possibilities to choose where to place key $x$.
- There are $m^{s-1}$ possibilities to choose the keys apart from $x$.
- There are $n^{s-1}$ possibilities to choose the cells.

Now, we analyze the probability that a phase is not successful without running into a closed cycle.

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{nm} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left( \frac{m}{n} \right)^s \leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left( \frac{1}{1+\epsilon} \right)^s \leq O \left( \frac{1}{m^2} \right).$$

Here we used the fact that $(1+\epsilon)m \leq n$.

Hence,

$$\Pr[\text{cycle}] = O \left( \frac{1}{m^2} \right).$$
Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with $x$ in the order that they are visited during the phase.

**Lemma 11**

If the sequence is of length $p$ then there exists a sub-sequence of at least $\frac{p+2}{3}$ keys starting with $x$ of **distinct** keys.

**Proof.**

Let $i$ be the number of keys (including $x$) that we see before the first repeated key. Let $j$ denote the total number of distinct keys.

The sequence is of the form:

$$x = x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_2, x_1 = x, x_8, x_9, \ldots$$

As $r \leq i - 1$ the length $p$ of the sequence is

$$p = i + r + (j - i) \leq i + j - 1.$$ 

Either sub-sequence $x_1 \rightarrow \cdots \rightarrow x_i$ or sub-sequence $x_1 \rightarrow x_{i+1} \rightarrow \cdots \rightarrow x_j$ has at least $\frac{p+2}{3}$ elements.
**Cuckoo Hashing**

A path-structure is active if for every key $x_ℓ$ (linking a cell $p_i$ from $T_1$ and a cell $p_j$ from $T_2$) we have 

$$h_1(x_ℓ) = p_i \text{ and } h_2(x_ℓ) = p_j$$

**Observation:**

If a phase takes at least $t$ steps without running into a cycle there must exist an active path-structure of size 

$$\left(\frac{2t + 2}{3}\right) \cdot \frac{n^2}{m^2}$$

Note that we count complete steps. A search that touches $2t$ or $2t + 1$ keys takes $t$ steps.

---

We choose $\text{maxsteps} \geq 3\ell/2 + 1/2$. Then the probability that a phase terminates unsuccessfully without running into a cycle is at most

$$\Pr[\text{unsuccessful} | \text{no cycle}] \leq 2\mu^2 \left(\frac{1}{1 + \epsilon}\right)^{2\ell + 1}$$

by choosing $\ell \geq \log \left(\frac{1}{2\mu^2 m^2}\right) / \log \left(\frac{1}{1 + \epsilon}\right) = \log (2\mu^2 m^2) / \log (1 + \epsilon)$

This gives $\text{maxsteps} = \Theta(\log m)$.

---

**Cuckoo Hashing**

The probability that a given path-structure of size $s$ is active is at most $\frac{n^2}{m^2}$.

The probability that there exists an active path-structure of size $s$ is at most

$$2 \cdot n^{s + 1} \cdot m^{s - 1} \cdot \frac{\mu^2}{n^2} \leq 2\mu^2 \left(\frac{m}{n}\right)^{s - 1} \leq 2\mu^2 \left(\frac{1}{1 + \epsilon}\right)^{s - 1}$$

Plugging in $s = (2t + 2)/3$ gives

$$\leq 2\mu^2 \left(\frac{1}{1 + \epsilon}\right)^{(2t+2)/3 - 1} = 2\mu^2 \left(\frac{1}{1 + \epsilon}\right)^{(2t-1)/3}.$$
Cuckoo Hashing
The expected number of complete steps in the successful phase of an insert operation is:

\[ E[\text{number of steps} \mid \text{phase successful}] = \sum_{t=1}^\infty \Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}] \]

We have

\[ \Pr[\text{search at least } t \text{ steps} \mid \text{successful}] = \frac{\Pr[\text{search at least } t \text{ steps} \land \text{successful}] \Pr[\text{no cycle}]}{\Pr[\text{successful}]} \leq \frac{1}{c} \frac{\Pr[\text{search at least } t \text{ steps} \land \text{no cycle}] \Pr[\text{no cycle}]}{\Pr[\text{no cycle}]} = \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] . \]

This means the expected cost for a successful phase is constant (even after accounting for the cost of the incomplete step that finishes the phase).

Cuckoo Hashing
Hence,

\[ E[\text{number of steps} \mid \text{phase successful}] \leq \frac{1}{c} \sum_{t=1}^\infty \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \]

\[ \leq \frac{1}{c} \sum_{t=1}^\infty 2\mu^2\left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3} = \frac{1}{c} \sum_{t=0}^\infty 2\mu^2\left(\frac{1}{1+\epsilon}\right)^{(2(t+1)-1)/3} \]

\[ = \frac{2\mu^2}{c(1+\epsilon)^{1/3}} \sum_{t=0}^\infty \left(\frac{1}{(1+\epsilon)^{2/3}}\right)^t = O(1) . \]

Formal Proof
Let \( Y_i \) denote the event that the \( i \)-th rehash does not lead to a valid configuration (assuming \( i \)-th rehash occurs) (i.e., one of the \( m + 1 \) insertions fails):

\[ \Pr[Y_i] \leq (m + 1) \cdot O(1/m^2) \leq O(1/m) =: p . \]

Let \( Z_i \) denote the event that the \( i \)-th rehash occurs:

\[ \Pr[Z_i] \leq \Pr[\bigwedge_{j=0}^{i-1} Y_j] \leq p^i \]

Let \( X_s^i, s \in \{1, \ldots, m + 1\} \) denote the cost for inserting the \( s \)-th element during the \( i \)-th rehash (assuming \( i \)-th rehash occurs):

\[ E[X_s^i] = E[\text{steps} \mid \text{phase successful}] \cdot \Pr[\text{phase successful}] + \text{maxsteps} \cdot \Pr[\text{not successful}] = O(1) . \]
The expected cost for all rehashes is

$$E\left[\sum_i \sum_s Z_i X_i^s\right]$$

Note that $Z_i$ is independent of $X_j^s$, $j \geq i$ (however, it is not independent of $X_j^s$, $j < i$). Hence,

$$E\left[\sum_i \sum_s Z_i X_i^s\right] = \sum_i \sum_s E[Z_i] \cdot E[X_i^s]$$

$$\leq O(m) \cdot \sum_i p_i^i$$

$$\leq O(m) \cdot \frac{p}{1-p}$$

$$= O(1) .$$

Cuckoo Hashing

What kind of hash-functions do we need?
Since $\text{maxsteps}$ is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.
Therefore, it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq (1 + \epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double $n$ and do a complete re-hash (table-expand).
- Whenever $m$ drops below $\alpha n/4$ we divide $n$ by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

Lemma 12

Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most $\frac{1}{2(1+\epsilon)}$.
Chapter 4 of [MS08] contains a detailed description about Hashing with Linear Probing and Hashing with Chaining. Also the Perfect Hashing scheme can be found there.

The analysis of Hashing with Chaining under the assumption of uniform hashing can be found in Chapter 11.2 of [CLRS90]. Chapter 11.3.3 describes Universal Hashing. Collision resolution with Open Addressing is described in Chapter 11.4. Chapter 11.5 describes the Perfect Hashing scheme.

Reference for Cuckoo Hashing???