Kapitel ADS:IV

IV. Datenstrukturen

- Record
- Linear List
- Linked List
- Stack
- Queue
- Priority Queue
- Dictionary
- Direct-address Table
- Hash Table
- Hash Function

Hash Function

Definition

Eine Hash Function (*Hashfunktion*)

$$h : U \to \{0, 1, \ldots, m-1\}$$

bildet ein Universum U von Schlüsseln beliebigen Typs auf m natürliche Zahlen ab.

Eigenschaften:

- total: Jeder Schlüssel k aus U hat genau einen Funktionswert $h(k)$ in $\{0, 1, \ldots, m-1\}$.
- surjektiv: Für alle $y \in \{0, 1, \ldots, m-1\}$ gibt es mindestens ein $k \in U$, so dass $h(k) = y$.

Hash Function

Definition

Eine Hash Function (*Hashfunktion*)

$$h : U \to \{0, 1, \ldots, m-1\}$$

bildet ein Universum U von Schlüsseln beliebigen Typs auf m natürliche Zahlen ab.

Eigenschaften:

- total: Jeder Schlüssel k aus U hat genau einen Funktionswert $h(k)$ in $\{0, 1, \ldots, m-1\}$.
- surjektiv: Für alle $y \in \{0, 1, \ldots, m-1\}$ gibt es mindestens ein $k \in U$, so dass $h(k) = y$.

Problemspezifikation

Problem: Hashing

Instanz: k. Ein Schlüssel aus U.

Lösung: i. Ein Wert aus $\{0, 1, \ldots, m-1\}$, der k deterministisch zugewiesen wird.

Wunsch: Eine Funktionsvorschrift oder ein Algorithmus, der das Hashingproblem für alle $k \in U$ so löst, dass Anwendungsanforderungen erfüllt werden.

Hash Function

Anwendungen

1. Dictionary / Mengen

   Ungeordnete Speicherung von Elementen unter einem eindeutigen Schlüssel bei Ausschluss von Duplikaten.

2. Ähnlichkeitssuche / Partitionierung

   Unterteilung von Elementen in Äquivalenzklassen, bestehend aus *ähnlichen* Elementen.

3. Datenintegritätstest / Kryptographie

   Sicherstellung der Echtheit einer Nachricht durch Abgleich mit einem Prüfwert.

Hash Function

Anwendungen

1. Dictionary / Mengen
   Ungeordnete Speicherung von Elementen unter einem eindeutigen Schlüssel bei Ausschluss von Duplikaten.

2. Ähnlichkeitssuche / Partitionierung
   Unterteilung von Elementen in Äquivalenzklassen, bestehend aus *ähnlichen* Elementen.

3. Datenintegritätstest / Kryptographie
   Sicherstellung der Echtheit einer Nachricht durch Abgleich mit einem Prüfwert.

Anforderungen gemäß Anwendung

1. Simple Uniform Hashing und Normalisierung
   Kollisionen sollen schlimmstenfalls zufällig gleichverteilt auftreten, unabhängig von der Verteilung der Schlüssel in U.

2. Ähnlichkeitssensitivität
   Kollisionen sollen genau dann auftreten, wenn sich zwei Schlüssel aus U ähnlich sind.

3. Kollisionsresistenz und Unumkehrbarkeit
   Kollisionen sollen mit an Sicherheit grenzender Wahrscheinlichkeit unmöglich sein. Aus Hashwerten sollen die ursprünglichen Schlüssel nicht rekonstruiert werden können.

Hash Function

Anwendungen

1. Dictionary / Mengen
   Ungeordnete Speicherung von Elementen unter einem eindeutigen Schlüssel bei Ausschluss von Duplikaten.

2. Ähnlichkeitssuche / Partitionierung
   Unterteilung von Elementen in Äquivalenzklassen, bestehend aus *ähnlichen* Elementen.

3. Datenintegritätstest / Kryptographie
   Sicherstellung der Echtheit einer Nachricht durch Abgleich mit einem Prüfwert.

Anforderungen gemäß Anwendung

1. Simple Uniform Hashing und Normalisierung
   Kollisionen sollen schlimmstenfalls zufällig gleichverteilt auftreten, unabhängig von der Verteilung der Schlüssel in U.

2. Ähnlichkeitssensitivität
   Kollisionen sollen genau dann auftreten, wenn sich zwei Schlüssel aus U ähnlich sind.

3. Kollisionsresistenz und Unumkehrbarkeit
   Kollisionen sollen mit an Sicherheit grenzender Wahrscheinlichkeit unmöglich sein. Aus Hashwerten sollen die ursprünglichen Schlüssel nicht rekonstruiert werden können.

Hash Function

Hash Tables

Problem: Hashing

Instanz: k. Ein Schlüssel aus U.

Lösung: i. Ein Wert aus $\{0, 1, \ldots, m-1\}$, der k deterministisch zugewiesen wird.

Wunsch: Eine Funktionsvorschrift oder ein Algorithmus, der das Hashingproblem für alle $k \in U$ gemäß Simple Uniform Hashing löst.

Hash Function
Hash Tables

Problem: Hashing

Instanz: k. Ein Schlüssel aus U.

Lösung: i. Ein Wert aus $\{0, 1, \ldots, m-1\}$, der k deterministisch zugewiesen wird.

Wunsch: Eine Funktionsvorschrift oder ein Algorithmus, der das Hashingproblem für alle $k \in U$ gemäß Simple Uniform Hashing löst.

Praktische Probleme:

- Das Universum der Schlüssel kann Schlüssel aller Datentypen enthalten.
- In der Praxis ist die Schlüsselverteilung unbekannt.

Hash Function

Hash Tables

Problem: Hashing

Instanz: k. Ein Schlüssel aus U.

Lösung: i. Ein Wert aus $\{0, 1, \ldots, m-1\}$, der k deterministisch zugewiesen wird.

Wunsch: Eine Funktionsvorschrift oder ein Algorithmus, der das Hashingproblem für alle $k \in U$ gemäß Simple Uniform Hashing löst.

Praktische Probleme:

- Das Universum der Schlüssel kann Schlüssel aller Datentypen enthalten.
- In der Praxis ist die Schlüsselverteilung unbekannt.

Heuristiken für Hashfunktionen:

- Divisionsrestmethode
- Multiplikative Methode
- Universelles Hashing

Hash Function

Vorverarbeitung

Annahme: Das Universum U kann auf die natürlichen Zahlen $\mathbf{N}$ abgebildet werden:

$$h : \mathbf{N} \rightarrow \{0, 1, \ldots, m-1\}$$

Die Abbildung ist abhängig vom Datentyp der Schlüssel in U.

Beispiel:

- Sei U die Menge aller Wörter und Schlüssel $k = \texttt{Turing}$ aus U.
- Zeichenketten (*Strings*) werden als Arrays von Zeichen repräsentiert.
- Zeichen sind auf Basis einer Kodierungstabelle als natürliche Zahlen kodiert.
- Jedem Zeichen ist ein Codepunkt in der Tabelle zugeordnet.
- Eine einfache Kodierungstabelle ist ASCII: sie kodiert 128 Zeichen.
- Zeichenketten können als Zahl zur Basis 128 kodiert werden:

$$k = \underbrace{84}_{\texttt{T}} \cdot 128^5 + \underbrace{117}_{\texttt{u}} \cdot 128^4 + \underbrace{114}_{\texttt{r}} \cdot 128^3 + \underbrace{105}_{\texttt{i}} \cdot 128^2 + \underbrace{110}_{\texttt{n}} \cdot 128^1 + \underbrace{103}_{\texttt{g}} \cdot 128^0$$

Hash Function

Vorverarbeitung

Annahme: Das Universum U kann auf die natürlichen Zahlen $\mathbf{N}$ abgebildet werden:

$$h : \mathbf{N} \rightarrow \{0, 1, \ldots, m-1\}$$

Die Abbildung ist abhängig vom Datentyp der Schlüssel in U.

Beispiel:

- Sei U die Menge aller Wörter und Schlüssel $k = \texttt{Turing}$ aus U.
- Zeichenketten (*Strings*) werden als Arrays von Zeichen repräsentiert.
- Zeichen sind auf Basis einer Kodierungstabelle als natürliche Zahlen kodiert.
- Jedem Zeichen ist ein Codepunkt in der Tabelle zugeordnet.
- Eine einfache Kodierungstabelle ist ASCII: sie kodiert 128 Zeichen.
- Zeichenketten können als Zahl zur Basis 128 kodiert werden:

$$k = 2.917.865.781.095_{10}$$

Bemerkungen:

- ASCII steht für „American Standard Code for Information Interchange“ und stellt einen frühen Standard zum Austausch von kodiertem Texten dar.

Hash Function

Divisionsrestmethode

Hashfunktion:

$$h(k) = k \bmod m,$$

wobei k ein Schlüssel aus U und m die Kapazität der Hash Table ist.

Beispiel: Aus $m = 12$ und $k = 100$ folgt $h(k) = 4$.

Hash Function

Divisionsrestmethode

Hashfunktion:

$$h(k) = k \bmod m,$$

wobei k ein Schlüssel aus U und m die Kapazität der Hash Table ist.

Beispiel: Aus $m = 12$ und $k = 100$ folgt $h(k) = 4$.

Eigenschaften:

- Sehr schnelle Berechnung; nur eine CPU-Instruktion.
- Die Kapazität m der Hash Table beeinflusst die Kollisionswahrscheinlichkeit:
  - Wenn m gerade ist, dann entspricht die Parität von $h(k)$ der von k.
  - Wenn $m = 2^p$, dann entspricht $h(k)$ nur den p niedrigstwertigen Bits.
  - Wenn $m = 2^p - 1$ (Mersenne-Zahl) und k ein String zur Basis 2^p, dann haben alle Permutationen einer Zeichenkette denselben Hashwert $h(k)$.

  ➜ Wenn m prim und stark verschieden von einer Zweierpotenz ist, verteilen sich die Hashwerte nahezu gleichmäßig.

Bemerkungen:

- Der Modulo-Operator mod (auch $\%$) ist eine Kurzform um die Division mit Rest auszudrücken. Für alle zwei ganzen Zahlen k und $m \neq 0$ gibt es zwei eindeutige ganze Zahlen a und b, so dass $k = ma + b$, wobei $0 \leq b < |m|$ für den Rest steht, der verbleibt, wenn man k durch m teilt.

Hash Function

Multiplikative Methode

Hashfunktion:

$$h(k) = \lfloor m(kc \bmod 1) \rfloor = \lfloor m(kc - \lfloor kc \rfloor) \rfloor,$$

wobei k ein Schlüssel aus U, m die Kapazität der Hash Table, und $0 < c < 1$ eine Konstante ist.

Hash Function

Multiplikative Methode

Hashfunktion:

$$h(k) = \lfloor m(kc \bmod 1) \rfloor = \lfloor m(kc - \lfloor kc \rfloor) \rfloor,$$

wobei k ein Schlüssel aus U, m die Kapazität der Hash Table, und $0 < c < 1$ eine Konstante ist.

Eigenschaften:

- Die Parameter m und c können unabhängig voneinander gewählt werden.
- Die Wahl von m ist unkritisch; wenn m eine Zweierpotenz ist, wird die Implementierung vereinfacht.
- Die Wahl von c beeinflusst die Kollisionswahrscheinlichkeit:
  - Wenn $c = (\sqrt{5} - 1)/2 = 0.6180339887...$ (Goldener Schnitt), verteilen sich die Hashwerte nahezu gleichmäßig.

Hash Function

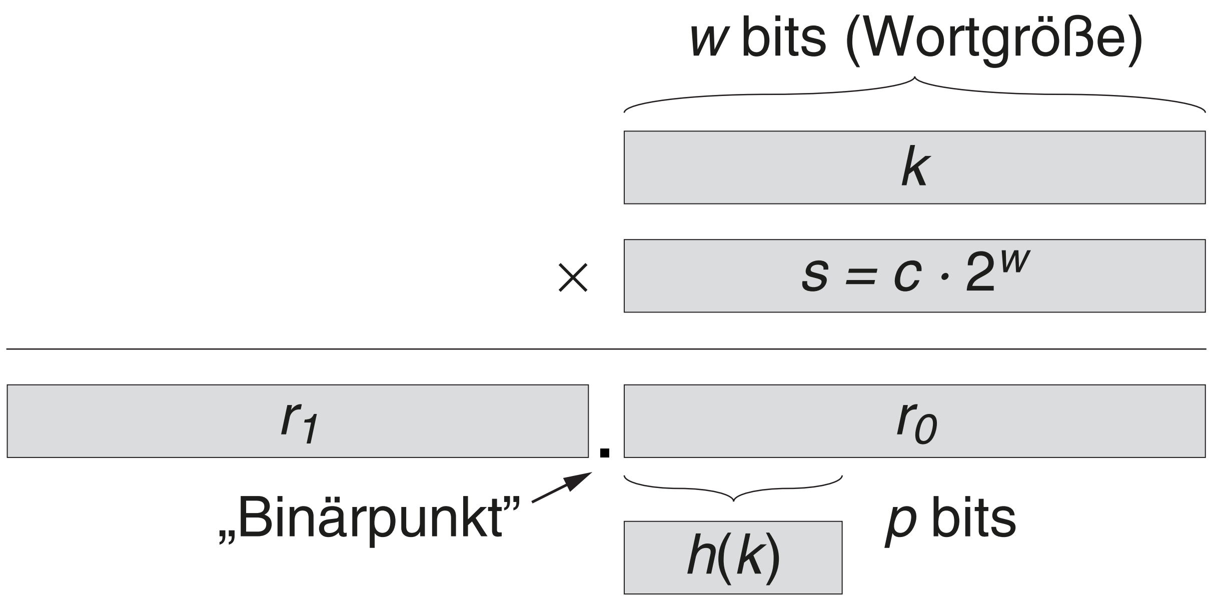
Multiplikative Methode

Hashfunktion:

$$h(k) = \lfloor m(kc \bmod 1) \rfloor = \lfloor m(kc - \lfloor kc \rfloor) \rfloor,$$

wobei k ein Schlüssel aus U, m die Kapazität der Hash Table, und $0 < c < 1$ eine Konstante ist.

Implementierung im Dualsystem für $m = 2^p$:



Hash Function
Multiplikative Methode

Hashfunktion:

$$h(k) = \lfloor m(kc \bmod 1) \rfloor = \lfloor m(kc - \lfloor kc \rfloor) \rfloor,$$

wobei k ein Schlüssel aus U, m die Kapazität der Hash Table, und $0 < c < 1$ eine Konstante ist.

Beispiel:

- Sei $m = 2^3 = 8$, $p = 3$, $w = 5$, und $k = 21$.

  Es muss $0 < s < 2^5$ gelten; wähle $s = 13$, so dass $c = 13/32$.

- Formelbasiert:

$$\begin{aligned} & kc = 21 \cdot \tfrac{13}{32} = \tfrac{273}{32} = 8\tfrac{17}{32} \\ \Rightarrow\ & kc \bmod 1 = \tfrac{17}{32} \\ \Rightarrow\ & m(kc \bmod 1) = 8\tfrac{17}{32} = \tfrac{17}{4} = 4\tfrac{1}{4} \\ \Rightarrow\ & \lfloor m(kc \bmod 1) \rfloor = 4 \\ \Rightarrow\ & h(k) = 4 \end{aligned}$$

- Implementierungsbasiert:

$$\begin{aligned} & ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \\ \Rightarrow\ & r_1 = 8, r_0 = 17 \\ \Rightarrow\ & r_0 = 10001_2 \\ \Rightarrow\ & h(k) = 100_2 \\ \Rightarrow\ & h(k) = 4 \end{aligned}$$

Hash Function
Universal Hashing

Gedankenspiel:

- Sei h die für eine Hash-Table-Implementierung festgelegte Hashfunktion.
- Dann kann ein böswilliger Nutzer (Adversary [*Gegenspieler*]) n Schlüssel aus U wählen, so dass alle ihre mit h berechneten Hashwerte kollidieren.

➜ Die Average-Case-Laufzeit kann nicht garantiert werden.

Gegenmaßnahme: Randomisierung

- Wähle zufällig eine andere Hashfunktion h vor jeder Nutzung.
- Solange der Adversary nicht vorhersagen kann, welche Funktion gewählt wird, kann die Average-Case-Laufzeit erwartet werden.

Probleme:

- Anzahl Funktionen von U nach m: $m^{|U|}$ ➜ $|U| \lg m$ bits pro Funktion.
- Zahlreiche mögliche Hashfunktionen haben nachteilige Eigenschaften.

➜ Konstruiere eine handhabbar große Familie von guten Hashfunktionen.

Hash Function

Universal Hashing: Definition

Sei H eine endliche Familie (Menge) von Hashfunktionen, die U auf $\{0, 1, \ldots, m-1\}$ abbilden. Wir nennen H c-universell, wenn für alle Schlüssel $k, l \in U$ die Zahl der Hashfunktionen $h \in H$, so dass $h(k) = h(l)$, höchstens $c/m \cdot |H|$ ist.

➜ Für ein zufälliges h aus H beträgt die Wahrscheinlichkeit c/m, dass $h(k) = h(l)$.

Satz 3 (Average-Case-Laufzeit III)

In einer Hash Table, in der Kollisionen mit Chaining behandelt und eine zufällige Hashfunktion aus einer Familie c-universeller Hashfunktionen verwendet wird, ist die Average-Case-Laufzeit bei erfolgreicher und erfolgloser Suche in $\Theta(1 + c\alpha)$.

Beweis: Analog zu Average-Case-Laufzeit I und II.

Der Hauptunterschied ist, dass der Analyse hier ein anderes Zufallsexperiment zugrundeliegt, nämlich das, eine Funktion h aus H zufällig zu wählen.

Hash Function

Universal Hashing: Hashfunktion I

Sei Hashfunktion $h_{\mathbf{a}}$ definiert als

$$h_{\mathbf{a}}(\mathbf{k}) = \mathbf{a}^T\mathbf{k} \bmod p,$$

wobei

- $\mathbf{a} = (a_1, \ldots, a_s)$ ein Vektor von Zufallszahlen mit $0 \leq a_i < p$,
- $\mathbf{k} = (k_1, \ldots, k_s)$ ein Vektor von Bestandteilen von Schlüssel k,
- $\mathbf{a}^T$ die Transposition von $\mathbf{a}$,
- $\mathbf{a}^T\mathbf{k} = \sum_{i=1}^{s} a_i k_i$ das Skalarprodukt der beiden Vektoren,
- und p eine Primzahl ist.

Hash Function

Universal Hashing: Hashfunktion I

Sei Hashfunktion $h_{\mathbf{a}}$ definiert als

$$h_{\mathbf{a}}(\mathbf{k}) = \mathbf{a}^T\mathbf{k} \bmod p,$$

wobei

- $\mathbf{a} = (a_1, \ldots, a_s)$ ein Vektor von Zufallszahlen mit $0 \leq a_i < p$,
- $\mathbf{k} = (k_1, \ldots, k_s)$ ein Vektor von Bestandteilen von Schlüssel k,
- $\mathbf{a}^T$ die Transposition von $\mathbf{a}$,
- $\mathbf{a}^T\mathbf{k} = \sum_{i=1}^{s} a_i k_i$ das Skalarprodukt der beiden Vektoren,
- und p eine Primzahl ist.

Beispiel:

$$\overbrace{\boxed{\quad\quad\quad\quad\quad\quad k \quad\quad\quad\quad\quad\quad}}^{w \text{ bits (Wortgröße)}}$$

Hash Function

Universal Hashing: Hashfunktion I

Sei Hashfunktion $h_{\mathbf{a}}$ definiert als

$$h_{\mathbf{a}}(\mathbf{k}) = \mathbf{a}^T\mathbf{k} \bmod p,$$

wobei

- $\mathbf{a} = (a_1, \ldots, a_s)$ ein Vektor von Zufallszahlen mit $0 \leq a_i < p$,
- $\mathbf{k} = (k_1, \ldots, k_s)$ ein Vektor von Bestandteilen von Schlüssel k,
- $\mathbf{a}^T$ die Transposition von $\mathbf{a}$,
- $\mathbf{a}^T\mathbf{k} = \sum_{i=1}^{s} a_i k_i$ das Skalarprodukt der beiden Vektoren,
- und p eine Primzahl ist.

Beispiel:

$$k_1 \quad k_2 \quad \ldots \quad \underbrace{k_s}_{\lfloor \lg p \rfloor \text{ bits}}$$

$\Rightarrow \; s = w / \lfloor \lg p \rfloor$

Hash Function

Universal Hashing: Hashfunktion I

Sei Hashfunktion $h_{\mathbf{a}}$ definiert als

$$h_{\mathbf{a}}(\mathbf{k}) = \mathbf{a}^T\mathbf{k} \bmod p,$$

wobei

- $\mathbf{a} = (a_1, \ldots, a_s)$ ein Vektor von Zufallszahlen mit $0 \leq a_i < p$,
- $\mathbf{k} = (k_1, \ldots, k_s)$ ein Vektor von Bestandteilen von Schlüssel k,
- $\mathbf{a}^T$ die Transposition von $\mathbf{a}$,
- $\mathbf{a}^T\mathbf{k} = \sum_{i=1}^{s} a_i k_i$ das Skalarprodukt der beiden Vektoren,
- und p eine Primzahl ist.

Beispiel:

	k_1	k_2	...	k_s	
	$\bullet$	$\bullet$	...	$\bullet$	
	a_1	a_2	...	a_s	
	=	=	...	=	
Σ	x_1	x_2	...	x_s	$= h(k)$

Hash Function

Universal Hashing: Hashfunktion I

Sei Hashfunktion $h_{\mathbf{a}}$ definiert als

$$h_{\mathbf{a}}(\mathbf{k}) = \mathbf{a}^T\mathbf{k} \bmod p,$$

wobei

- $\mathbf{a} = (a_1, \ldots, a_s)$ ein Vektor von Zufallszahlen mit $0 \leq a_i < p$,
- $\mathbf{k} = (k_1, \ldots, k_s)$ ein Vektor von Bestandteilen von Schlüssel k,
- $\mathbf{a}^T$ die Transposition von $\mathbf{a}$,
- $\mathbf{a}^T\mathbf{k} = \sum_{i=1}^{s} a_i k_i$ das Skalarprodukt der beiden Vektoren,
- und p eine Primzahl ist.

Satz 4 (Universelle Hashfunktionen I)

Die Familie von Hashfunktionen

$$H_1 = \{h_{\mathbf{a}} \mid \mathbf{a} \in \{0, 1, \ldots, p-1\}^s\}$$

ist 1-universell, wenn p eine Primzahl ist.

Beweisidee: Abschätzung der Wahrscheinlichkeit, dass $h(\mathbf{k}_1) = h(\mathbf{k}_2)$.

Hash Function

Universal Hashing: Hashfunktion II

Sei Hashfunktion $h_{a,b}$ definiert als

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$$

wobei

- $a \in \{1, 2, \ldots, p-1\} = \mathbf{Z}_p^*$,
- $b \in \{0, 1, \ldots, p-1\} = \mathbf{Z}_p$,
- p eine Primzahl,
- und $m < p$ die Kapazität der Hash Table.

Satz 5 (Universelle Hashfunktionen II)

Die Familie von Hashfunktionen

$$H_2 = \{h_{a,b} \mid a \in \mathbf{Z}_p^* \text{ and } b \in \mathbf{Z}_p\}$$

ist 1-universell, wenn p eine Primzahl ist.

Beweisidee: Abschätzung der Wahrscheinlichkeit, dass $h(k_1) = h(k_2)$.

Bemerkungen:

- Für jede Zahl $\alpha > 1$ und jede nicht zu kleine natürliche Zahl m enthält das Intervall $[m, \alpha m]$ etwa $(\alpha - 1)m / \ln m$ Primzahlen. Es genügt also für häufig genutzte Intervalle, Tabellen mit eine Reihe von Primzahlen bereitzustellen. Auch die Suche nach einer Primzahl in einem Intervall ist möglich.

Hash Function

Perfect Hashing

Voraussetzung:

- Die Menge $K \subseteq U$ tatsächlich benötigter Schlüssel ist vollständig bekannt.
- K ist statisch; es werden weder Elemente hinzugefügt noch gelöscht.

➜ Wunsch: Vermeidung von Hashkollisionen.

Hash Function

Perfect Hashing

Voraussetzung:

- Die Menge $K \subseteq U$ tatsächlich benötigter Schlüssel ist vollständig bekannt.
- K ist statisch; es werden weder Elemente hinzugefügt noch gelöscht.

➜ Wunsch: Vermeidung von Hashkollisionen.

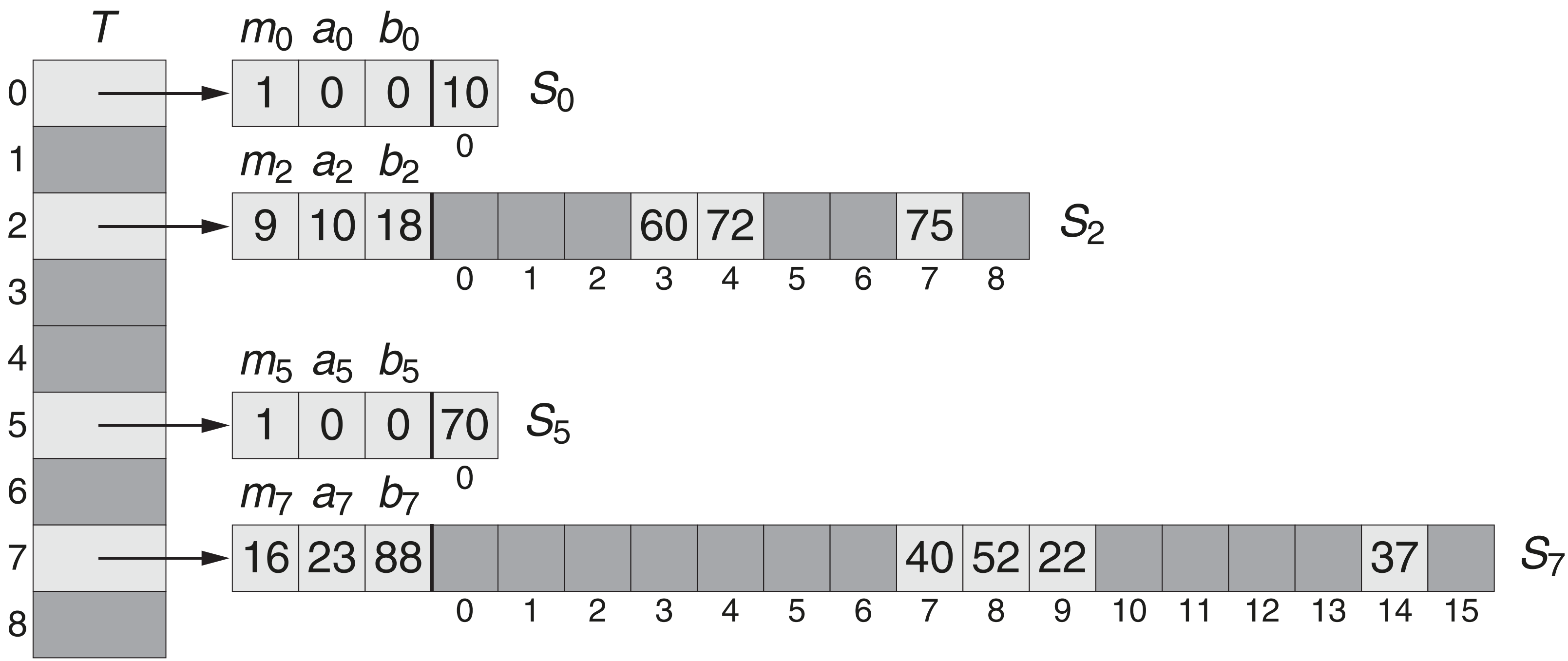
Konstruktion

- Sei H eine Familie universeller Hashfunktionen.
- Verteilung der n Schlüssel auf die $m = n = |K|$ Slots und $h \in H$.
- Wenn der i-te Slot $n_i > 0$ Elemente erhält: Wähle solange ein $h_i \in H$, bis die Schlüssel kollisionsfrei auf $m_i = n_i^2$ Slots verteilen.

Hash Function

Perfect Hashing

Beispiel für $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$ und $h_{3,42}$, $p = 101$ und $m = 9$:



Wenn $m_j = n_j = 1$ genügt die Hashfunktion mit $a = b = 0$

Hash Function

Perfect Hashing: Analyse

Satz 6 (Perfekte Hashfunktionen I)

Wenn n Schlüssel mit einer Hashfunktion h, die zufällig aus einer Familie universeller Hashfunktionen gezogen wurde, auf $m = n^2$ Slots verteilt werden, ist die Wahrscheinlichkeit für eine Hashkollision kleiner als $1/2$.

Beweisidee: Abschätzung der erwarteten Zahl von Kollisionen für n Schlüssel unter universellem Hashing bei n^2 möglichen Slots.

Hash Function

Perfect Hashing: Analyse

Satz 6 (Perfekte Hashfunktionen I)

Wenn n Schlüssel mit einer Hashfunktion h, die zufällig aus einer Familie universeller Hashfunktionen gezogen wurde, auf $m = n^2$ Slots verteilt werden, ist die Wahrscheinlichkeit für eine Hashkollision kleiner als $1/2$.

Beweisidee: Abschätzung der erwarteten Zahl von Kollisionen für n Schlüssel unter universellem Hashing bei n^2 möglichen Slots.

Satz 7 (Perfekte Hashfunktionen II)

Wenn n Schlüssel mit einer Hashfunktion h, die zufällig aus einer Familie universeller Hashfunktionen gezogen wurde, auf $m = n$ Slots verteilt werden, und die Größe der sekundären Hash Tables $m_i = n_i^2$ für $i = 0, 1, \ldots, m-1$, dann ist die Wahrscheinlichkeit, dass der kumulierte Platzverbrauch der sekundären Hash Tables $4n$ übersteigt, kleiner als $1/2$.

Beweisidee: Abschätzung der erwarteten Summe der benötigten Kapazitäten n_j^2 für alle $j = 0, 1, \ldots, m-1$ benötigten sekundären Hash Tables.