Kapitel ADS:IV

IV. Datenstrukturen

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

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:

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

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:

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

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

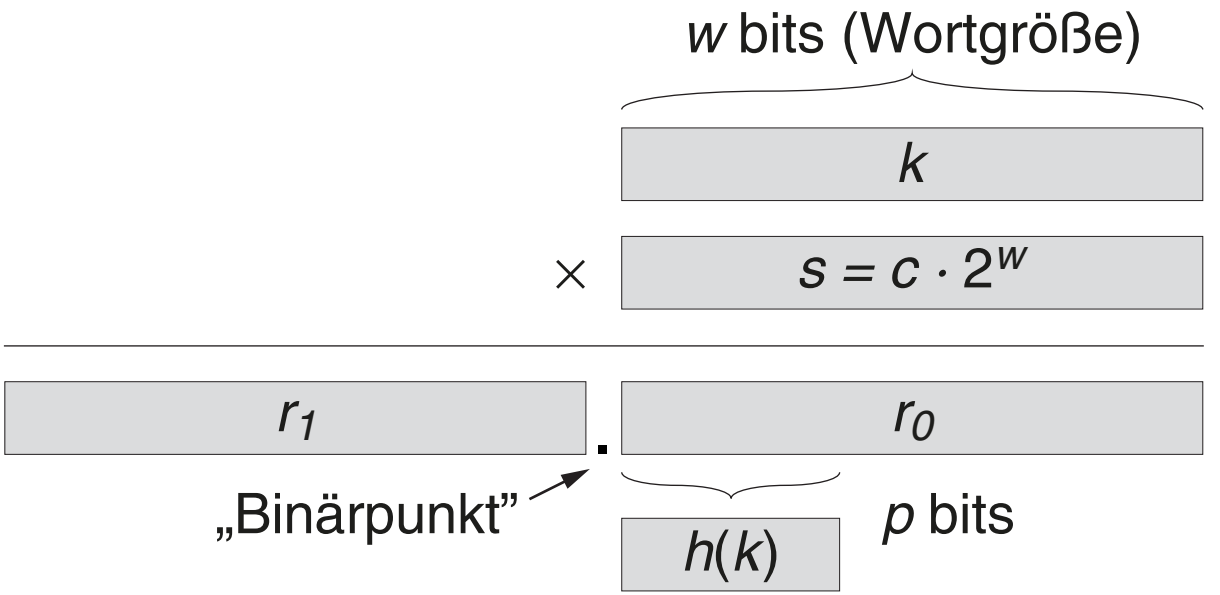
Multiplikative Methode

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:

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