13 Hashing

Wörterbuchoperationen:

- $S.\,\textbf{insert}(x)$: Füge Element x.
- $S.\,\textbf{delete}(x)$: Lösche das durch x referenzierte Element.
- $S.\,\textbf{search}(k)$: Gib eine Referenz auf Element e zurück mit $\text{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

13 Hashing

Wörterbuchoperationen:

- $S.\mathbf{insert}(x)$: Füge Element x.
- $S.\mathbf{delete}(x)$: Lösche das durch x referenzierte Element.
- $S.\mathbf{search}(k)$: Gib eine Referenz auf Element e zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

13 Hashing

Wörterbuchoperationen:

- $S.\mathbf{insert}(x)$: Füge Element x.
- $S.\mathbf{delete}(x)$: Lösche das durch x referenzierte Element.
- $S.\mathbf{search}(k)$: Gib eine Referenz auf Element e zurück mit $\text{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

13 Hashing

Wörterbuchoperationen:

- $\boldsymbol{S.\,\mathrm{insert}(x)}$: Füge Element x.
- $\boldsymbol{S.\,\mathrm{delete}(x)}$: Lösche das durch x referenzierte Element.
- $\boldsymbol{S.\,\mathrm{search}(k)}$: Gib eine Referenz auf Element e zurück mit $\mathrm{key}[e] = k$ falls existent; andernfalls gib NULL zurück.

Suchbäume unterstützen diese Operationen mit Laufzeit $\mathcal{O}(\log n)$. Es werden Vergleichselemente ausgewählt.

Dann wird eine Objekt gesucht indem schrittweise mit diesen Elementen verglichen wird.

Hashing versucht direkt den Speicherort des jeweiligen Objektes zu berechnen. Das Ziel ist eine konstante Suchzeit.

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \le |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein.
- gut zu speichern (klein).
- eine gute Verteilung der Elemente über die Hashtabelle garantieren.

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein.
- gut zu speichern (klein).
- eine gute Verteilung der Elemente über die Hashtabelle garantieren.

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \dots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \dots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

13 Hashing

Definitionen:

- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \dots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \dots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

13 Hashing

Definitionen:

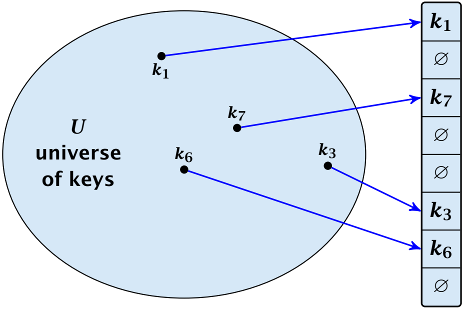
- Universum U von Schlüsseln, e.g., $U \subseteq \mathbb{N}_0$. U sehr groß.
- Teilmenge $S \subseteq U$ von Schlüsseln, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ Hashtabelle.
- Hashfunktion $h : U \to [0, \ldots, n-1]$.

Die Hashfunktion h sollte:

- schnell auswertbar sein
- gut zu speichern (klein)
- eine gute Verteilung der Elemente über die Hashtabelle garantieren

Direkte Addressierung

Idealerweise bildet die Hashfunktion alle Elemente auf unterschiedliche Tabelleneinträge ab.



Diesen Spezialfall nennt man Direkte Adressierung. Selten möglich, da das Universum üblicherweise zu groß ist.

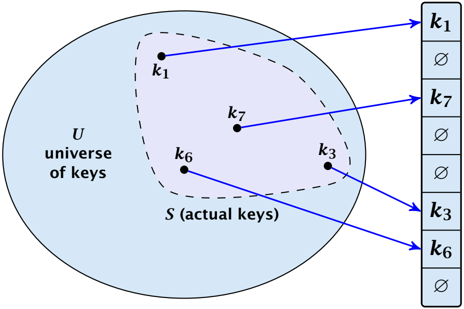
Direkte Adressierung

Operationen

- `insert(x)`
  `A[x->key]=x`
- `search(k)`
  `return A[k]`
- `delete(x)`
  `A[x->key]=NULL`

Perfektes Hashing

Angenommen wir kennen die Menge S der auftretenen Schlüssel (kein Löschen/ kein Einfügen). Dann möchte man eine einfache Hashfunktion finden, die alle Schlüssel auf unterschiedliche Positionen abbildet.



Solche eine Hashfunktion h nennt man eine perfekte Hashfunktion für Menge S.

Perfektes Hashing

Operationen

- `insert(x)`
  `A[h(x->key)]=x`
- `search(k)`
  `return A[h(k)]`
- `delete(x)`
  `A[h(x->key)]=NULL`

Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

Problem: Kollisionen
Üblicherweise ist das Universum U viele größer als die Tabellengröße n.

Zwei Elemente k_1, k_2 aus S können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

Problem: Kollisionen

Üblicherweise ist das Universum U viele größer als die Tabellengröße n.

Zwei Elemente k_1, k_2 aus S können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

Kollisionen

Falls wir die Schlüssel nicht kennen, ist das beste, dass die Hashfunktion diese gleichmäßig über die Tabelle verteilt.

Problem: Kollisionen
Üblicherweise ist das Universum U viele größer als die Tabellengröße n.

Zwei Elemente k_1, k_2 aus S können auf den gleichen Speicherort abbilden (d.h., $h(k_1) = h(k_2)$). Dies nennt man eine Hashkollision.

Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente S sich $\Theta(\sqrt{n})$ nähert.

Lemma 1

Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn m Elemente in eine Tabelle der Größe n abgebildet werden ist mindestens

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

Uniformes Hashing:

Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \ldots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente S sich $\Theta(\sqrt{n})$ nähert.

Lemma 1

Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn m Elemente in eine Tabelle der Größe n abgebildet werden ist mindestens

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

Uniformes Hashing:

Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \dots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

Kollisionen

Typischerweise treten Kollisionen auf wenn die Anzahl der Elemente S sich $\Theta(\sqrt{n})$ nähert.

Lemma 1

Die Wahrscheinlichkeit einer Kollision bei uniformem Hashing wenn m Elemente in eine Tabelle der Größe n abgebildet werden ist mindestens

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

Uniformes Hashing:

Die Hashfunktion ist eine zufällige Funktion aus der Menge aller Funktionen $f : U \to [0, \dots, n-1]$.

Uniformes Hashing ist nicht praktikabel, da solch eine Hashfunktion sehr viel Speicherplatz benötigt.

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\Pr[A_{m,n}]$$

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n}$$

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

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

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\left(1-\frac{j}{n}\right)\\ &\leq \prod_{j=0}^{m-1} e^{-j/n}\end{aligned}$$

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\left(1-\frac{j}{n}\right)\\ &\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}}\end{aligned}$$

Kollisionen

Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\Big(1-\frac{j}{n}\Big) \\ &\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \enspace .\end{aligned}$$

Kollisionen

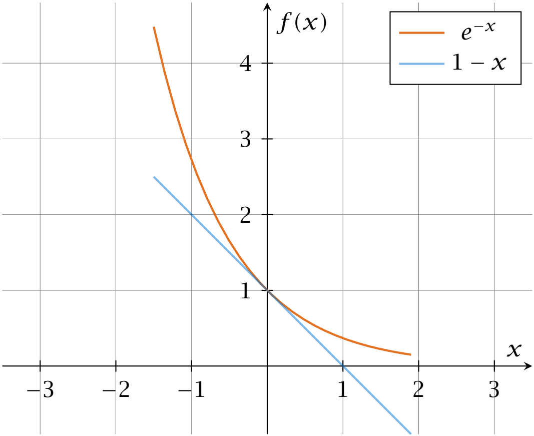
Beweis.

Sei $A_{m,n}$ das Ereignis dass das Einfügen von m Schlüsseln in eine Tabelle der Größe n keine Kollision verursacht. Dann

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\left(1-\frac{j}{n}\right) \\ &\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \enspace .\end{aligned}$$

Die erste Gleichung folgt, da das ℓ-te Element das gehashed wird mit Wahrscheinlichkeit $\frac{n-\ell+1}{n}$ keine Kollision verursacht (unter der Bedingung, das die vorherigen Elemente nicht kollidiert sind). □

Kollisionen



Die Ungleichung $1 - x \leq e^{-x}$ erhält man wenn man die Taylorexpansion von e^{-x} nach dem 2. Term abbricht.

Kollisionsauflösung

Es gibt zwei Hauptarten der Kollisionsauflösung

- Offen Adressierung (auch bekannt unter „open addressing“, „closed hashing“)
- Hashing mit Verkettung (auch bekannt unter „hashing with chaining“, „closed addressing“, „open hashing“).

Es gibt Anwendungen (z.B. Computerschach) wo Kollisionen nicht aufgelöst werden.

Kollisionsauflösung

Es gibt zwei Hauptarten der Kollisionsauflösung

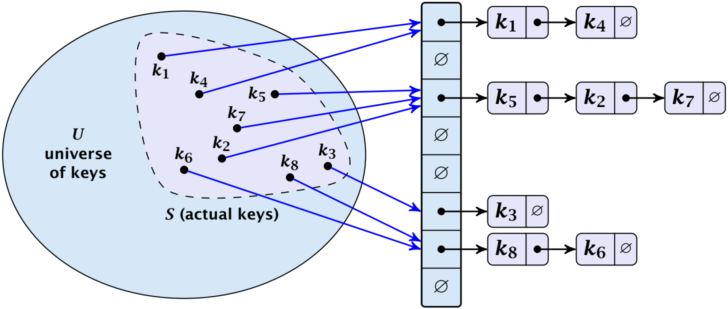
- Offen Adressierung (auch bekannt unter „open addressing“, „closed hashing“)
- Hashing mit Verkettung (auch bekannt unter „hashing with chaining“, „closed addressing“, „open hashing“).

Es gibt Anwendungen (z.B. Computerschach) wo Kollisionen nicht aufgelöst werden.

Hashing mit Verkettung

Füge Elemente, die auf die gleiche Position abgebildet werden in verkettete Liste ein.

- Search: berechne $h(x)$ und durchsuche Liste nach `x->key`.
- Insert: Einfügen am Anfang der Liste; $\mathcal{O}(1)$
- Delete: doppelt verkette Liste: $\mathcal{O}(1)$
  einfach verkette Liste: suchen + $\mathcal{O}(1)$



Hashing mit Verkettung

Im worst-case werden alle Elemente auf eine Liste gemapped.

Laufzeit dann $\Omega(n)$ für search.

sehr schlecht

Auswege:

- average-case Analyse
- Randomisierung; bestimme erwartete Laufzeit für die Wahl einer zufälligen Hashfunktion aus einer Menge von Hashfunktionen.

Hashing mit Verkettung

Uniformes Hashing: wähle zufällige Hashfunktion aus Menge aller Funktionen

Lemma

Falls m Elemente mittels uniformem Hashing in Hashtabelle der Größe n gespeichert werden dann ist die erwartete Laufzeit einer search-Operation $\mathcal{O}(1 + m/n)$.

Speicherverbrauch für Hashfunktion zu groß!

Beweis

- führe `search(k)` aus;
- erwartete Laufzeit ist $\mathcal{O}(1 + E[X])$, wobei X Zufallsvariable für Länge der Liste $A[h(k)]$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes Element; $X_e = 1 \Leftrightarrow h(\texttt{e->key}) = h(k)$
- Listenlänge $X = \sum_e X_e$
- erwartete Listenlänge:

$$E[\textstyle\sum_{e\in S} X_e] = \displaystyle\sum_{e\in S} E[X_e] = \sum_{e\in S} \Pr[X_e = 1] = \sum_{e\in S} 1/n = m/n$$

c-universelles Hashing

Definition

Eine Familie $\mathcal{H}$ von Hashfunktionen auf $\{0, \dots, n-1\}$ heißt c-universell falls für jedes Schlüsselpaar k_1, k_2 gilt, dass

$$|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| \leq \frac{c}{n}|\mathcal{H}|$$

D.h. bei zufälliger Wahl der Hashfunktion gilt

$$\Pr[h(k_1) = h(k_2)] \leq \frac{c}{n}$$

c-universelles Hashing

Lemma

Falls m Elemente mittels einer zufälligen Hashfunktion aus einer c-universellen Klasse in Hashtabelle der Größe n gespeichert werden dann ist die erwartete Laufzeit einer search-Operation $\mathcal{O}(1 + cm/n)$.

Beweis

- $X_e \in \{0,1\}$; $X_e = 1 \Leftrightarrow h(\texttt{e->key} = h(k))$
- Listenlänge für Schlüssel k ist: $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$
\begin{aligned}
E[X] &= \sum[\sum_{e\in S} X_e] = \sum_{e\in S} E[X_e] \\
&= \sum_{e\in S} \Pr[X_e = 1] \le \sum_{e\in S} c/n = cm/n
\end{aligned}
$$

Hashing mit Verkettung

Nachteile:

- Zeiger erhöhen Speicherverbrauch
- schlechte Cache-Effizienz

Vorteile:

- kein festes Limit für die Anzahl der Elemente
- Löschen kann effizient implementiert werden

Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im j-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von 0,...,$n-1$ sein.

Search(k): Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ...

Insert(x): Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im j-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von 0,...,$n-1$ sein.

$\mathbf{Search}(k)$: Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ...

$\mathbf{Insert}(x)$: Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im j-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von 0,...,$n-1$ sein.

Search(k): Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ...

Insert(x): Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im j-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von 0,...,$n-1$ sein.

Search(k): Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ...

Insert(x): Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

Offene Adressierung

Alle Objekte werden in der Tabelle gespeichert.

Funktion $h(k,j)$ definiert Tabellenposition, die im j-ten Schritt untersucht wird. Werte $h(k,0)$,...,$h(k,n-1)$ muss Permutation von 0,...,$n-1$ sein.

Search(k): Versuche Position $h(k,0)$; falls leer ist Element nicht vorhanden; sonst versuche $h(k,1)$, $h(k,2)$, ...

Insert(x): Suche bis zu einem leeren Tabellenplatz; dort wird das Element eingefügt. Falls die Suche bis $h(k,n-1)$ geht (und der Platz nicht leer ist) ist die Tabelle voll.

Offene Adressierung

Möglichkeiten für $h(k,j)$:

- Lineares Sondieren:
  $h(k,i) = h(k) + i \mod n$
  (manchmal: $h(k,i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k,i) = h_1(k) + i h_2(k) \mod n$.

Offene Adressierung

Möglichkeiten für $h(k, j)$:

- Lineares Sondieren:
  $h(k, i) = h(k) + i \mod n$
  (manchmal: $h(k, i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k, i) = h_1(k) + ih_2(k) \mod n$.

Offene Adressierung

Möglichkeiten für $h(k, j)$:

- Lineares Sondieren:
  $h(k, i) = h(k) + i \mod n$
  (manchmal: $h(k, i) = h(k) + ci \mod n$).
- Quadratisches Sondieren:
  $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Doppeltes Hashing:
  $h(k, i) = h_1(k) + ih_2(k) \mod n$.

Offen Adressierung

Nachteile:

- nachträgliche Änderung der Tabellengröße sehr aufwändig
- Löschen schwierig

Vorteile:

- Lineares Sondierung ist Cache-effizient;
- keine Zeiger; geringerer Speicherverbrauch