Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Hashing

- Einführung
- Kollisionsauflösung durch Verkettung
- Hashfunktionen
- Offene Adressierung

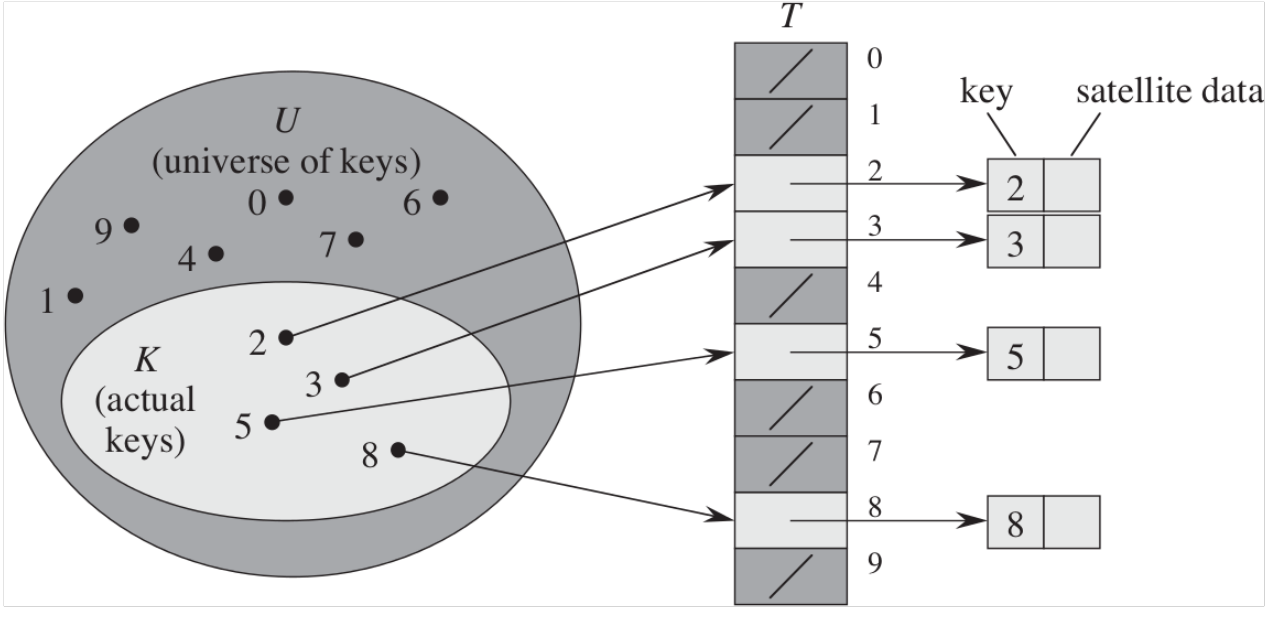
Einführung

- Datenstruktur zur effizienten Implementation der Wörterbuchoperationen
  - Insert, Search, Delete
  - Elemente enthalten Schlüssel und Satellitendaten
- Einfachste Lösung: Feld
  - Schlüssel werden als Indizes verwendet
  - Direkter Zugriff: Gegeben Schlüssel k, gesuchtes Element steht an Position k im Feld
  - Brauchen gleich viele Elemente im Feld wie mögliche Schlüssel!

Direkter Zugriff

- Ziel: Dynamische Menge mit Wörterbuch-operationen
- Annahme: Schlüssel aus Universum $U = \{0, \dots, m-1\}$
  - m nicht zu gross
  - Keine doppelten Schlüssel
- Adresstabelle mit direktem Zugriff ist Feld $T[0, \dots, m-1]$
  - Falls Element mit Schlüssel k existiert, dann enthält $T[k]$ Zeiger auf Element
  - Sonst $T[k] = \text{NIL}$

Direkter Zugriff



DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

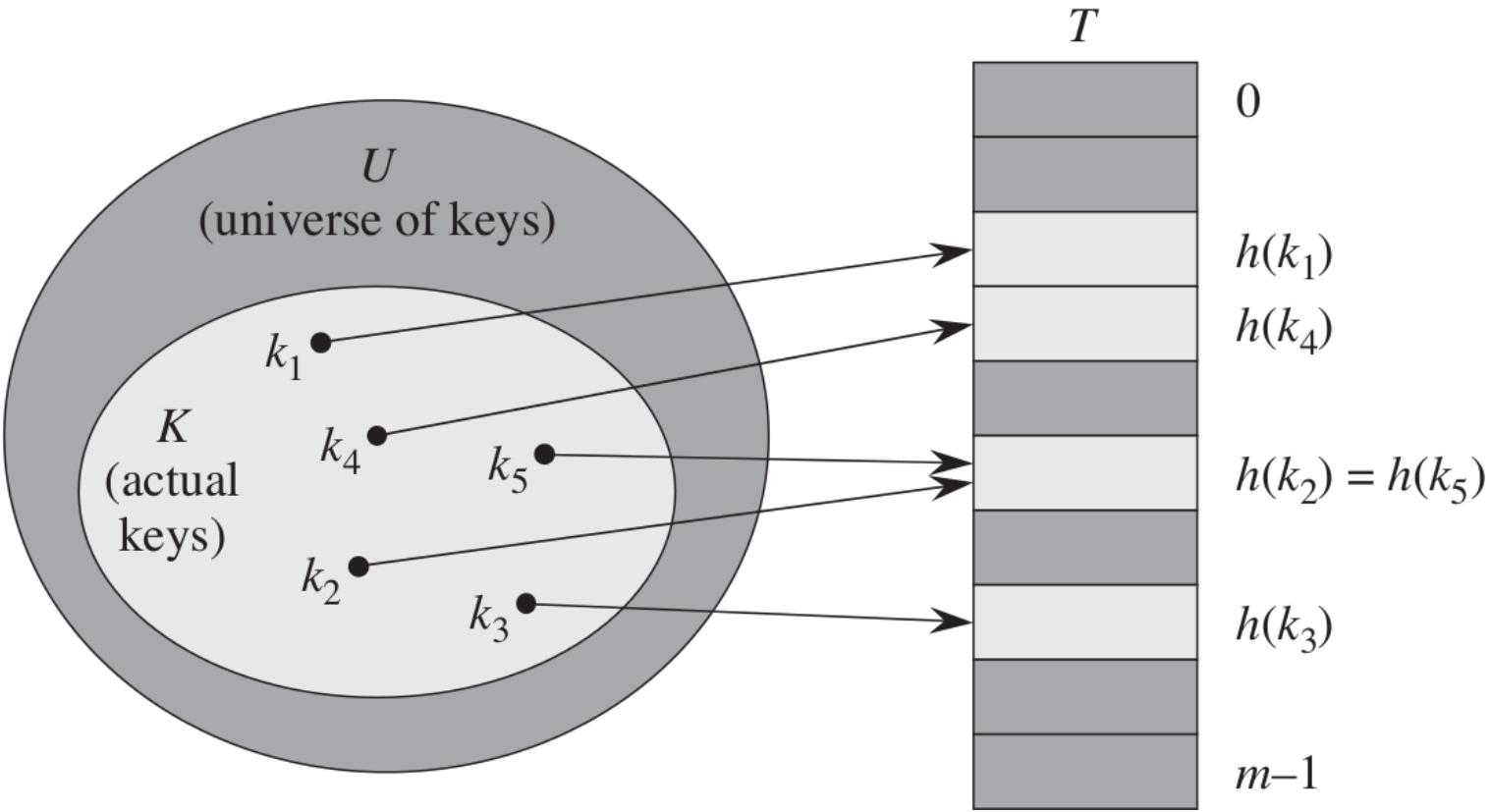
Hashtabellen

- Nützlich wenn Anzahl möglicher Schlüssel viel grösser als Anzahl gespeicherter Elemente
  - Direkter Zugriff vergeudet Speicher
- Grösse der Hashtabelle ist normalerweise
  - proportional zur Anzahl gespeicherter Elemente
  - nicht Anzahl möglicher Schlüssel

Hashtabellen

- Hauptidee
  - Gegeben Schlüssel k, berechne Index $h(k)$ in Hashtabelle in Abhängigkeit von k
- Themen
  - Hashfunktionen: Aus Schlüssel Index in Hashtabelle berechnen: $k \rightarrow h(k)$
  - Kollisionen: zwei Schlüssel ergeben selben Index: $h(k_1) = h(k_2)$

Hashtabellen



Hashfunktion h

Hashtabellen

- Nützlich, wenn Anzahl $|K|$ gespeicherter Schlüssel viel kleiner als Anzahl $|U|$ möglicher Schlüssel ist
- Benutze Feld der Grösse $m = \Theta(|K|)$
  - Speicherbedarf $\Theta(|K|)$ statt $\Theta(|U|)$
- Idee: Hashfunktion h
  - Schlüssel k wird an Index $h(k)$ gespeichert

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

- Zeitkomplexität von Suchen
  - Average case $O(1)$, leider nicht worst-case

Kollisionen

- Zwei oder mehr Schlüssel ergeben denselben Hashwert
  - Nicht auszuschliessen da Schlüsseluniversum viel grösser als Hashtabelle, d.h. $|U| > m$
- Zwei Methoden zur Behandlung von Kollisionen
  - Verkettung
  - Offene Adressierung

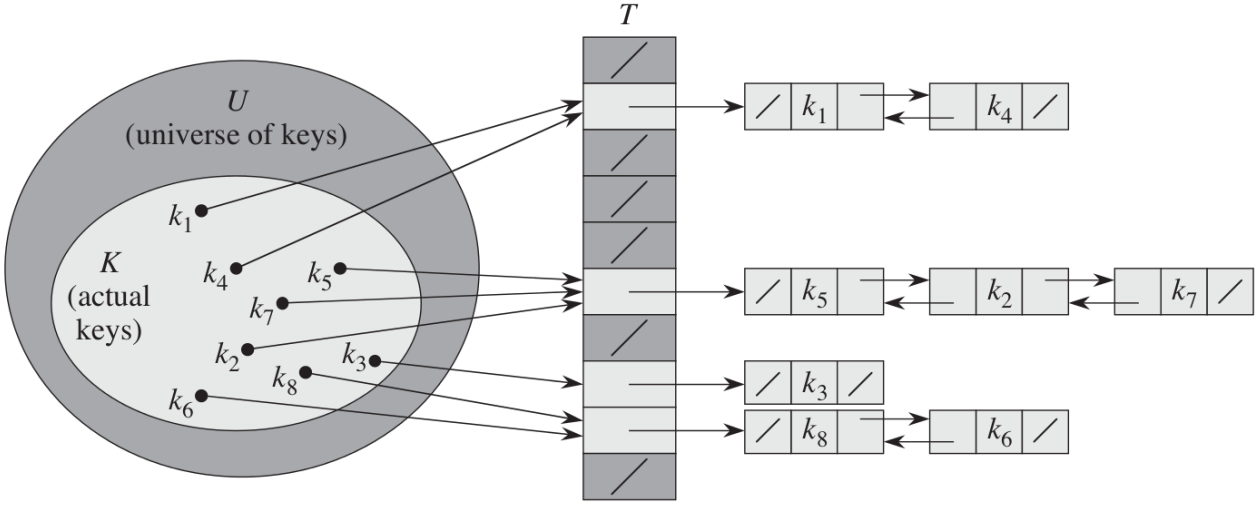
Übersicht

Hashing

- Einführung
- Kollisionsauflösung durch Verkettung
- Hashfunktionen
- Offene Adressierung

Verkettung

- Hashtabelle speichert Zeiger auf Listen
- Listen enthalten Elemente mit gleichem Hashwert



- Einfache Verkettung auch möglich

Einfügen

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

- Worst-case Laufzeit $O(1)$
  - Unter Annahme, dass Schlüssel nicht schon existiert in Hashtabelle
  - Sonst muss zusätzlich mittels Suchen geprüft werden, ob der Schlüssel schon in der Liste vorkommt

Löschen

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

- Keine Suche nötig wenn Zeiger auf Element x gegeben ist
- Worst-case Laufzeit mit doppelt verketteten Listen ist $O(1)$
- Einfach verkettete Listen erfordern Traversierung wie beim Suchen
  - Brauchen Vorgänger von x um dessen $next$ Zeiger anzupassen

Suchen

Analyse

- Gegeben Schlüssel, was ist der Aufwand um Element mit dem Schlüssel zu finden, oder zu entscheiden, dass Schlüssel nicht existiert?
- Abhängig von Belegungsfaktor $\alpha = n/m$
  - Anzahl Elemente in Hashtabelle n
  - Grösse der Hashtabelle m
  - Durchschnittliche Anzahl Elemente pro Liste

$$\alpha < 1, \alpha = 1, \text{oder } \alpha > 1$$

Analyse: Worst case

- Alle n Schlüssel belegen den gleichen Slot
- Eine Liste der Länge n
- Worst-case Suche dauert $\Theta(n)$

Analyse: Average case

- Annahme: einfaches gleichmässiges Hashing
  - Jedes Element wird mit gleicher Wahrscheinlichkeit auf jeden der m Slots gehasht
- Notation
  - Länge der Liste $T[j]$ ist n_j
  - Anzahl Element in Tabelle $n = n_0 + n_1 + \cdots + n_{m-1}$
  - Durchschnitt von n_j ist $E[n_j] = n/m = \alpha$
- Annahme: Hashfunktion h berechnet in konstanter Zeit

Analyse: Average case

- Zwei Fälle
  - Erfolgreiche Suche: Hash Tabelle enthält gesuchten Schlüssel
  - Erfolglose Suche: Hash Tabelle enthält gesuchten Schlüssel nicht
- Theorem
  Erfolglose Suche ist im Mittel $\Theta(1 + \alpha)$

Erfolglose Suche: Beweisidee

- Einfaches gleichmässiges Hashing: jeder Schlüssel k der noch nicht in der Tabelle ist, wird mit gleicher Wahrscheinlichkeit auf jeden der m Plätze abgebildet
- Erfolgloses Suchen: müssen ganze Liste $T[h(k)]$ durchsuchen
  - Erwartete Länge ist $E[n_{h(k)}] = \alpha$
  - Erwartete Anzahl besuchter Elemente ist α
  - Aufwand inkl. Berechnung der Hash-funktion ist $\Theta(1 + \alpha)$

Erfolgreiche Suche

- Achtung
  - Erfolglose Suche: alle Listen werden mit gleicher Wahrscheinlichkeit durchsucht
- Anders bei erfolgreicher Suche
  - Wahrscheinlichkeit, dass gewisse Liste durchsucht wird ist proportional zur Anzahl Elemente in dieser Liste
- Theorem
  Erfolgreiche Suche ist im Mittel $\Theta(1 + \alpha)$

Erfolgreiche Suche: Beweisidee

- Annahme: jedes der n Elemente in der Hashtabelle ist mit gleicher Wahrscheinlichkeit das gesuchte Element x
- Anzahl durchsuchter Elemente ist eines mehr als Anzahl Elemente vor x in der Liste, die x enthält
  - Elemente, die nach x eingefügt wurden (einfügen am Kopf der Liste)
- Brauchen Durchschnitt für Anzahl Elemente die nach x in die Liste von x eingefügt wurden

Erfolgreiche Suche: Beweisidee

Notation

- x_i ist i-tes Element, das in Tabelle eingefügt wurde
- Schlüssel $k_i = x_i.key$
- Indikator-Zufallsvariable für Kollision
$$X_{ij} = I\{h(k_i) = h(k_j)\}$$
- Wahrscheinlichkeit für Kollision bei einfachem gleichmässigen Hashing
$$\Pr\{h(k_i) = h(k_j)\} = 1/m \Rightarrow E[X_{ij}] = 1/m$$

Erfolgreiche Suche: Beweisidee

\# Schlüssel, die vor Schlüssel k_i in Liste eingefügt wurden: $\sum_{j=i+1}^{n} X_{ij}$

$\Longrightarrow$ \# Schritte um Schlüssel k_i zu finden: $1 + \sum_{j=i+1}^{n} X_{ij}$

Durchschnitt über alle n eingefügte Schlüssel: $\frac{1}{n} \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} X_{ij} \right)$

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

Erfolgreiche Suche: Beweisidee

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

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)=\frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{nm}\sum_{i=1}^{n}\sum_{j=i+1}^{n}1=$$

$$1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)=$$

$$1+\frac{1}{mn}\left(n^2-\frac{n\cdot(n+1)}{2}\right)=1+\frac{n-1}{2m}=$$

$$1+\frac{\alpha}{2}-\frac{\alpha}{2n}=\Theta(1+\alpha)$$

Übersicht

Hashing

- Einführung
- Kollisionsauflösung durch Verkettung
- Hashfunktionen
- Offene Adressierung

Hashfunktionen

- Idealfall: Hashfunktion führt zu einfachem gleichmässigem Hashing
  - Jedes Element wird mit gleicher Wahrscheinlichkeit auf irgendeinen Platz der Tabelle abgebildet
  - Unabhängig von anderen Schlüsseln
- In der Praxis nicht möglich
  - Verteilung der Schlüssel unbekannt
  - Schlüssel nicht unabhängig
- Heuristische Methoden
  - Hashwert soll nicht von Mustern beeinflusst werden, die möglicherweise in Daten existieren
  - „to hash“ = „zerhacken“

Divisionsmethode

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

- Beispiel: $m = 12, k = 100 \Rightarrow h(k) = 4$
- Vorteil: schnell berechnen, eine Division
- Nachteil: gewisse Werte von m sind ungünstig
  - Zweierpotenzen: wenn $m = 2^p$ dann entspricht $h(k)$ den niederwertigste p bits von k
- Gute Wahl: Primzahl nicht zu nahe bei Zweierpotenz

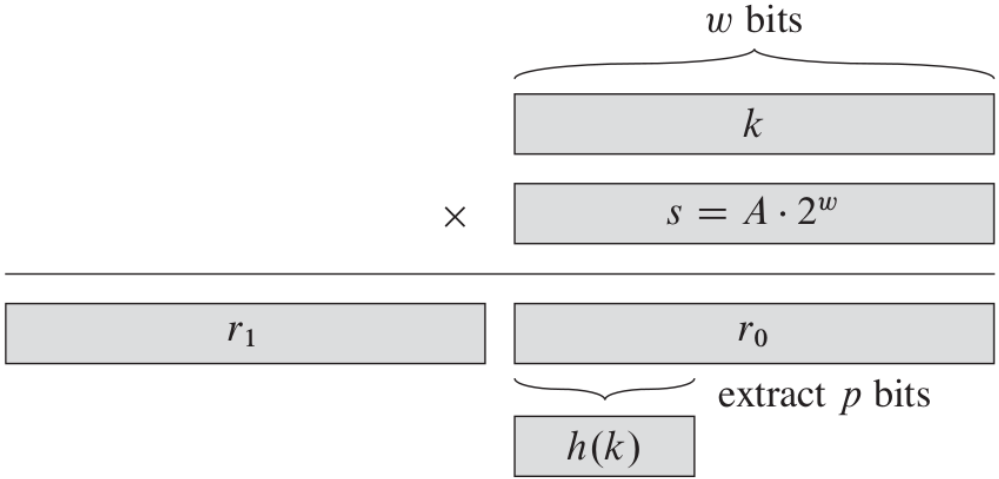
Multiplikationsmethode

$$h(k) = \lfloor m(k\ A \bmod 1) \rfloor$$

- Rezept
  - Wähle Konstante A: $0 < A < 1$
  1. Multipliziere Schlüssel k mit A
  2. Extrahiere Nachkommastellen mittels „$\bmod 1$“
  3. Multipliziere mit m
  4. Runde ab
- Nachteil: langsamer als Divisionsmethode
- Vorteil: Wert von m nicht kritisch

Effiziente Implementation

- Datenwort habe w bits (typisch $w \in \{32, 64\}$)
- Schlüssel k brauchen w bits
- Wähle
  - Grösse der Hashtabelle Zweierpotenz $m = 2^p$
  - Integer s, $0 < s < 2^w$, und $A = {}^{s}/_{2^w}$



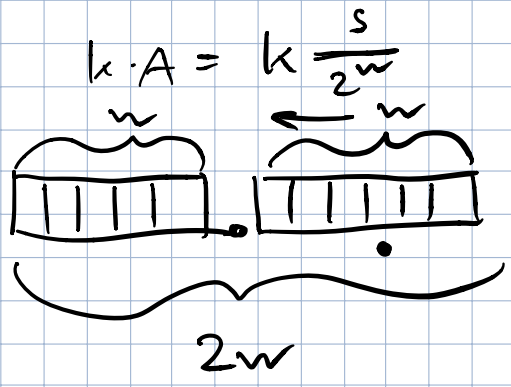
$1 = s/2^w$

$k = \lfloor m(kA \bmod 1) \rfloor$

$0 < s < 2^w$

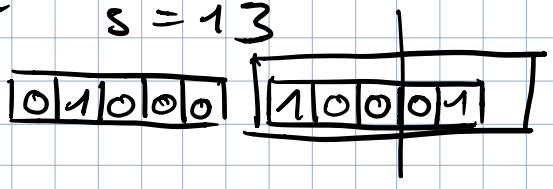
$k \cdot s$

$k \cdot A = k \frac{s}{2^w}$



$p = 3 \quad (m = 2^p = 8) \quad w = 5 \quad s = 13$

$k = 21 \qquad s \cdot k = 273$



$(100)_2 = 4$

Also Hashwert von 21 bei gewählten Parametern ist 4.

Beispiel

- Parameter $p = 3$ $(m = 8), w = 5, s = 13$
- Schlüssel $k = 21$

siehe oben

Übersicht

Hashing

- Einführung
- Kollisionsauflösung durch Verkettung
- Hashfunktionen
- Offene Adressierung

Offene Adressierung

- Alternative zu Verkettung für Kollisionsbehandlung
- Idee: speichere alle Schlüssel in Hashtabelle selbst
  - Jeder Platz enthält einen Schlüssel oder NIL
- Hashtabelle kann voll werden
  - Belegungsfaktor immer $\alpha \leq 1$
- Kollisionsbehandlung: berechne Sequenz von Plätzen, die sondiert werden

Hashfunktion

$$h : U \times \underbrace{\{0,1,\ldots,m-1\}}_{\text{Sondierung}} \rightarrow \underbrace{\{0,1,\ldots,m-1\}}_{\text{Platz}}$$

- Sequenz der sondierten Plätze muss Permutation von $(0,1,\ldots,m-1)$ sein
  - Anders formuliert: Sondierungssequenz
    $$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$$
    muss Permutation von $(0,1,\ldots,m-1)$ sein
  - Jeder Platz muss genau einmal sondiert werden

Suchen

- Suchen Schlüssel k
- Algorithmus

  Initialisiere Sondierung $i = 0$

  1. Berechne $h(k, i)$
  2. Falls $h(k, i)$ Schlüssel k enthält: erfolgreiche Suche
  3. Falls $h(k, i)$ NIL enthält: erfolglose Suche
  4. Falls $h(k, i)$ anderen Schlüssel enthält: inkrementiere i ($i \leftarrow i + 1$)
  5. Falls $i = m$: erfolglose Suche, sonst: → 1.

Suchen & Einfügen

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == k
5            return j
6       i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3       j = h(k, i)
4       if T[j] == NIL
5            T[j] = k
6            return j
7       else i = i + 1
8  until i == m
9  error "hash table overflow"
```

Löschen

- Einfach NIL an den gelöschten Platz schreiben funktioniert nicht!
- Warum?

Löschen

Lösung

- Verwende zusätzlichen Wert deleted anstatt NIL um anzuzeigen, dass Element aus Platz gelöscht wurde
- Suchen behandelt deleted wie wenn ein Schlüssel gespeichert würde, der nicht mit gesuchtem Schlüssel übereinstimmt
- Einfügen behandelt deleted wie freien Platz
- Nachteil
  - Aufwand für Suchen hängt nicht mehr vom Belegungsfaktor ab

Sondierungssequenzen

- Was sind die Eigenschaften einer idealen Methode, um Sondierungssequenzen zu erzeugen?
- Praktische Verfahren, um Sondierungssequenzen zu erzeugen

Sondierungssequenzen

- Idealfall: gleichmässiges Hashing
  Jeder Schlüssel hat mit gleicher Wahrscheinlichkeit irgendeine der $m!$ Permutation von $(0,1,\ldots,m-1)$ als Sondierungssequenz
  - Verallgemeinerung von einfachem gleichmässigem Hashing
- Gleichmässiges Hashing in der Praxis nicht möglich
  - Garantiere zumindest dass Sondierungssequenz eine Permutation von $(0,1,\ldots,m-1)$ ist
  - Keine der folgenden Techniken erzeugt alle $m!$ Permutationen

Lineares Sondieren

- Hilfshashfunktion h'
  - z.B. Multiplikationsmethode
- Lineares Sondieren hat Hashfunktion

$$h(k,i) = (h'(k) + i\ ) \bmod m$$

  - Sondierung i
- Erzeugt Sondierungssequenz

$$T[h'(k)], T[h'(k)+1], \dots, T[m-1], T[0], \dots, T[h'(k)-1]$$

- Nur m verschiedene Sequenzen!

Lineares Sondieren

Nachteil: Primäres Clustering

- Lange Folgen besetzter Plätze
  - Leerer Platz folgend auf i besetzte Plätze wird mit Wahrscheinlichkeit $(i + 1)/m$ als nächstes belegt
  - Lange Folgen besetzter Plätze werden noch länger
  - Suchen wird aufwändiger

Quadratisches Sondieren

- Hashfunktion

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

  - Hilfskonstanten $c_1, c_2 \neq 0$
- Besser als lineares Sondieren
  - Kein primäres Clustering
- Nur m Sondierungssequenzen
  - Alle Schlüssel k mit gleicher ersten Sondierung $h(k,0)$ führen zu gleicher Sequenz
  - Sekundäres Clustering

Doppeltes Hashing

- Zwei Hilfshashfunktionen h_1, h_2
- Hashfunktion

$$h(k,i) = \left(h_1(k) + i\, h_2(k)\right) \bmod m$$

- Bedingung: $h_2(k)$ ist teilerfremd zu m damit Sondierungssequenz eine Permutation von $(0,1,\ldots,m-1)$ erzeugt
  - Keine gemeinsamen Faktoren ausser 1
- Mögliche Lösungen
  - m ist Zweierpotenz und h_2 ist immer ungerade
  - m ist prim und $1 < h_2(k) < m$

Doppeltes Hashing

- Vorteil gegenüber linearem und quadratischem Sondieren
  - Erzeugt m^2 statt m verschiedene Sondierungssequenzen
  - Jede Kombination von $h_1(k)$ und $h_2(k)$ ergibt andere Sequenz
- Verhalten in der Praxis nahe am Idealfall des gleichmässigen Hashing

Analyse von offenem Hashing

Annahmen

- Analyse in Bezug auf Belegungsfaktor α
- Tabelle nie komplett voll, d.h. $0 \leq \alpha < 1$
- Idealfall: gleichmässiges Hashing
- Kein Entfernen von Schlüsseln
- Suche nach jedem Schlüssel in Tabelle gleich wahrscheinlich

Analyse von erfolgloser Suche

- Theorem: erwartete Anzahl Sondierungen bei erfolgloser Suche ist höchstens $^1/_{1-\alpha}$

Zufallsvariable X = Anzahl Sondierungen bei erfolgloser Suche

Gesucht: erwartete Anzahl Sondierungen $E[X]$

Ereignis A_i: es gibt eine i-te Probe auf einen besetzten Platz

$X \geq i$ bedeutet: es werden Sondierungen $1, 2, \dots, i-1$ auf besetzte Plätze gemacht.

$$\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$$
$$= \Pr\{A_1\} \cdot \Pr\{A_2|A_1\} \cdot \Pr\{A_3|A_1 \cap A_2\} \dots$$
$$\Pr\{A_{i-1}|A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$

Beweis

Beh: $\Pr\{A_j | A_1 \cap \cdots \cap A_{j-1}\} = \frac{n-j+1}{m-j+1}$

- $j = 1$: n gespeicherte Schlüssel, m Plätze, W'keit besetzten Platz zu sondieren: $\frac{n}{m}$
- Sonst: $j - 1$ Sondierungen auf besetzte Plätze bereits erfolgt
  gleichmässiges Hashing → nächster sondierter Platz wurde bisher noch nicht sondiert.
  Anzahl verbl. Plätze: $m - (j-1) = m - j + 1$
  davon belegt: $n - (j-1) = n - j + 1$
  → W'keit dass j-te Probe auf belegten Platz fällt: $\frac{n-j+1}{m-j+1}$

Beweis

- → $\Pr\{X \geq 1\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-1+2}{m-1+2}}_{i-1 \text{ Faktoren}}$
- Weil $n < m$: $\frac{n-j}{m-j} \leq \frac{n}{m}$

  → $\Pr\{X \geq i\} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$

Beweis

- Erwartungswert:

$$
\begin{aligned}
E[X] &= \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\} && \text{| Gleichung } (C.25) \\
&= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
&\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\
&= \sum_{i=0}^{\infty} \alpha^{i} && \text{| geom. Reihe} \\
&= \frac{1}{1-\alpha}
\end{aligned}
$$

Analyse von Einfügen

- Einfügen erfordert den gleichen Ablauf wie erfolglose Suche
- Folgerung: Einfügen mit offener Adressierung bei Belegungsfaktor α unter Annahme von gleichmässigem Hashing erfordert im Mittel $^{1}/_{1-\alpha}$ Sondierungen

Analyse von erfolgreicher Suche

- Theorem: Erwartete Anzahl Sondierungen in erfolgreicher Suche ist höchstens $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
  - Erfolgreiche Suche nach k erzeugt dieselbe Sondierungssequenz wie beim Einfügen von k
  - Wenn k als $(i+1)$-ter Schlüssel eingefügt wird, dann hatte α beim Einfügen den Wert $\frac{i}{m}$
  - Die erwartete Anzahl Sondierungen ist deshalb

$$\frac{1}{1-\alpha} = \frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$$

Beweis

Alle Schlüssel → Durchschnitt

$$\frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i}$$

$$= \frac{1}{\alpha} \cdot \sum_{k=m-n+1}^{m} \frac{1}{k} \qquad \text{| Ungl. } (A.12)$$

$$\leq \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x} dx$$

$$= 1/\alpha \cdot (\ln m - \ln(m-n))$$

$$= 1/\alpha \cdot \ln(m/(m-n))$$

$$= \frac{1}{\alpha} \cdot \ln \frac{1}{1 - {}^{n}/_{m}}$$

$$= \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$$

Zusammenfassung

- Wörterbuchoperationen
  $\mathrm{Insert},\ \mathrm{Search},\ \mathrm{Delete}$
- Hashtabelle $T[0,1,\ldots,m-1]$
  - Anzahl Elemente $\ll$ Anzahl möglicher Schlüssel
  - Hashfunktion h: $U \to \{0,\ldots,m-1\}$
  - Kollision: $h(k_1) = h(k_2)$
    → Verkettung oder offene Adressierung

Zusammenfasung

- Verkettung
  - $T[h(k)]$ enthält verkettete Liste von Schlüsseln
  - Belegungsfaktor $\alpha > 1$ möglich
  - Einfügen: $O(1)$
  - Löschen: $O(1)$, unter gew. Voraussetzungen
  - Suchen:
    - Worst-case: $\Theta(n)$
    - Average-case: $\Theta(1 + \alpha)$

Zusammenfassung

- Offene Adressierung
  - $T[h(k)]$ enthält Schlüssel
  - Belegungsfaktor $\alpha \leq 1$
  - Suche freie Plätze durch Sondierung $h(k, i)$
  - Idealfall: gleichmässiges Hashing
    Erwartete Anzahl Sondierungen:
    - Einfügen und erfolglose Suche: $\leq \frac{1}{1-\alpha}$
    - Erfolgreiche Suche: $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Zusammenfassung

- Sondierungsmethoden
  - lineares Sondieren: $h(k,i) = (h'(k) + i) \bmod m$
  - quadratisches Sondieren:
    $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - doppeltes Hashing: $h(k,i) = \left(h_1(k) + i h_2(k)\right) \bmod m$
    wobei $h_2(k)$ teilerfremd zu m
- Hashfunktionen
  - Divisionsmethode: $h(k) = k \bmod m$
    schnell, gewisse Werte von m ungünstig
  - Multiplikationsmethode: $h(k) = \lfloor m(k\,A \bmod 1) \rfloor$
    Wert von m nicht kritisch,
    langsamer als Divisionsmethode

Nächstes Mal

- Kapitel 12: Binäre Suchbäume