Kapitel ADS:IV

IV. Datenstrukturen

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

Dictionary

Definition

Ein Datenstruktur heißt Dictionary („*Wörterbuch*“), wenn eingefügte Elemente anhand ihres Schlüssels gefunden und gelöscht werden können.

Dictionary

Definition

Ein Datenstruktur heißt Dictionary („*Wörterbuch*“), wenn eingefügte Elemente anhand ihres Schlüssels gefunden und gelöscht werden können.

Manipulation

- Element einfügen (*Insert*)
  Ein Element in das Dictionary einfügen.
- Element suchen (*Search*)
  Ein Element anhand seines Schlüssels im Dictionary nachschlagen.
- Element löschen (*Delete*)
  Ein Element aus dem Dictionary löschen.

Dictionary

Definition

Ein Datenstruktur heißt Dictionary („*Wörterbuch*"), wenn eingefügte Elemente anhand ihres Schlüssels gefunden und gelöscht werden können.

Manipulation

- Element einfügen (*Insert*)
  Ein Element in das Dictionary einfügen.
- Element suchen (*Search*)
  Ein Element anhand seines Schlüssels im Dictionary nachschlagen.
- Element löschen (*Delete*)
  Ein Element aus dem Dictionary löschen.

Implementierung

- Direct-address Table
  Verwendung der Schlüssel als Indexe eines Arrays.
- Hash Table
  Verwendung von „Hashing" zur Abbildung der Schlüsselmenge auf die Indexe eines Arrays.
- Suchbaum
  Verwendung von Baum-basierten Datenstrukturen.

Bemerkungen:

- Alternativ wird das Dictionary auch Associative Array (*assoziatives Array*) oder Map (*Abbildung*) genannt.
- Wir nehmen an, dass Elemente paarweise verschiedene Schlüssel haben.

Direct-address Table

Definition

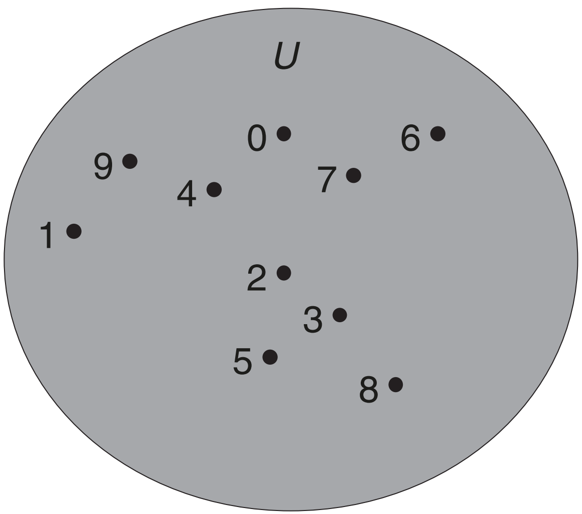
Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Beispiel:



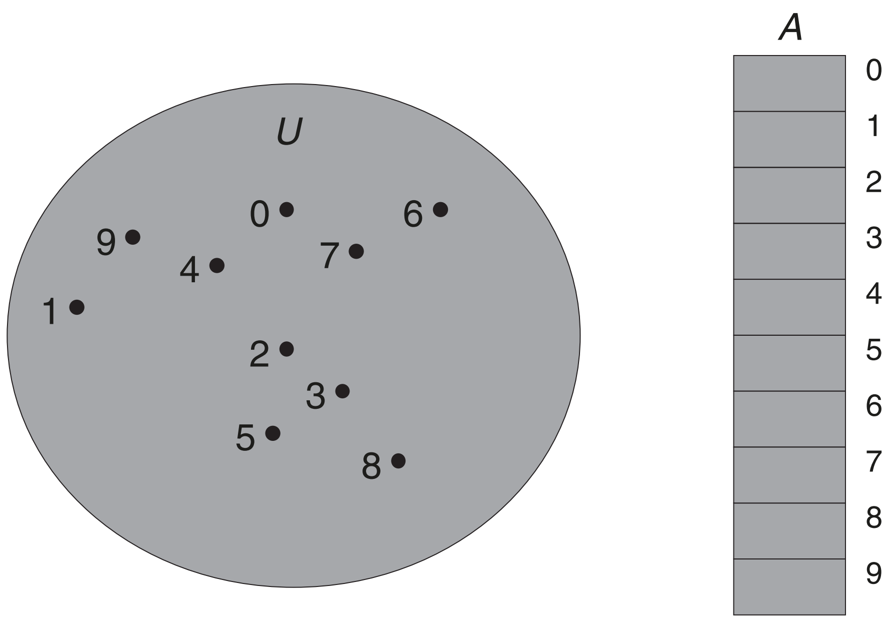
$U = \{0, 1, \ldots, m-1\}$ ist das Universum möglicher Schlüssel

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Beispiel:



$U = \{0, 1, \ldots, m-1\}$ ist das Universum möglicher Schlüssel

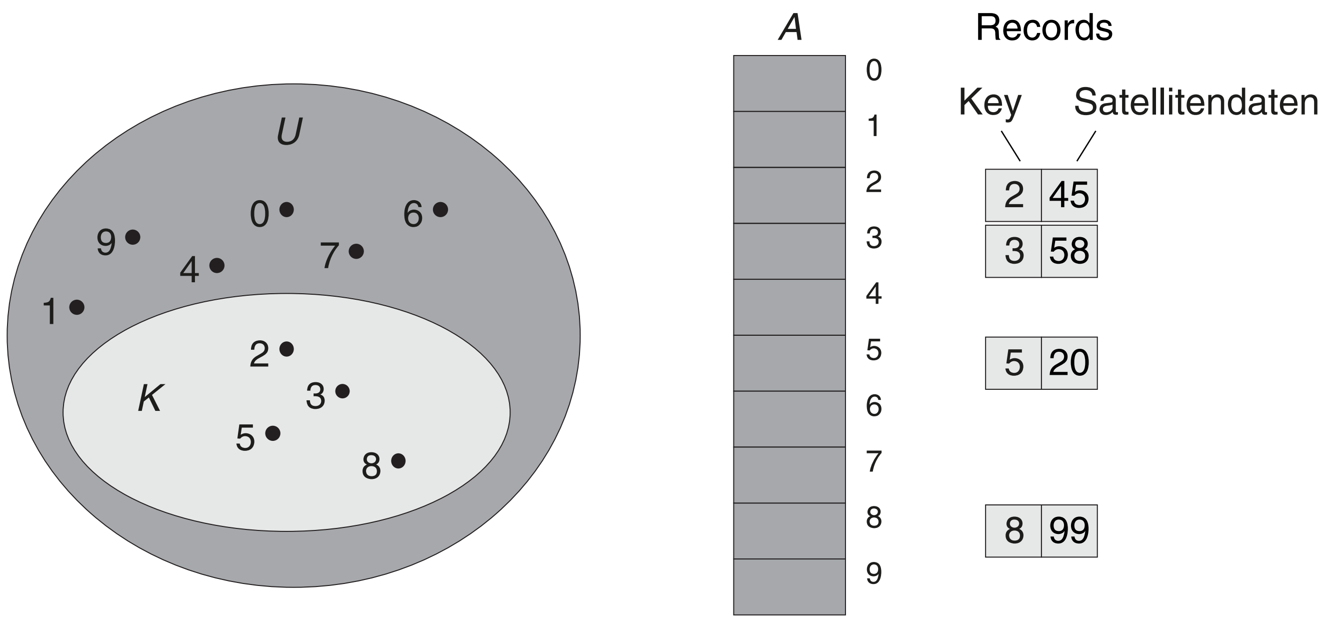
A ist ein 0-indiziertes Array der Länge m

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Beispiel:



$U = \{0, 1, \ldots, m-1\}$ ist das Universum möglicher Schlüssel

A ist ein 0-indiziertes Array der Länge m

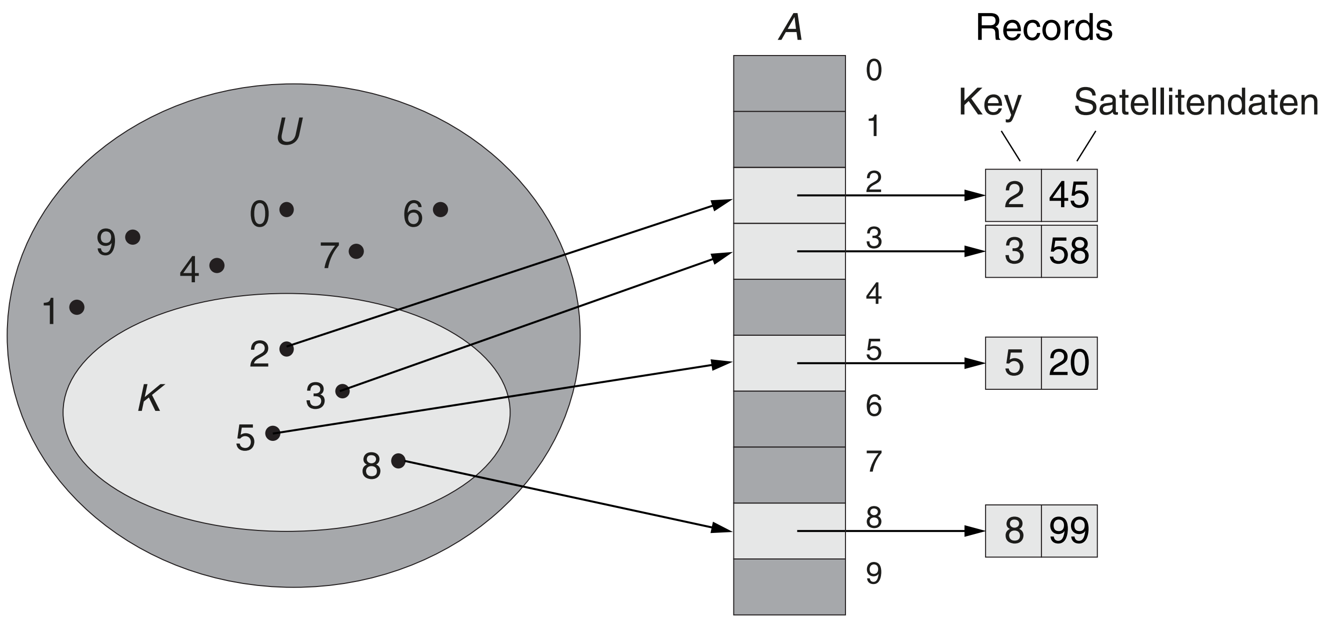
$K \subseteq U$ ist die Teilmenge tatsächlich benutzter Schlüssel

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Beispiel:



$U = \{0, 1, \ldots, m-1\}$ ist das Universum möglicher Schlüssel

A ist ein 0-indiziertes Array der Länge m

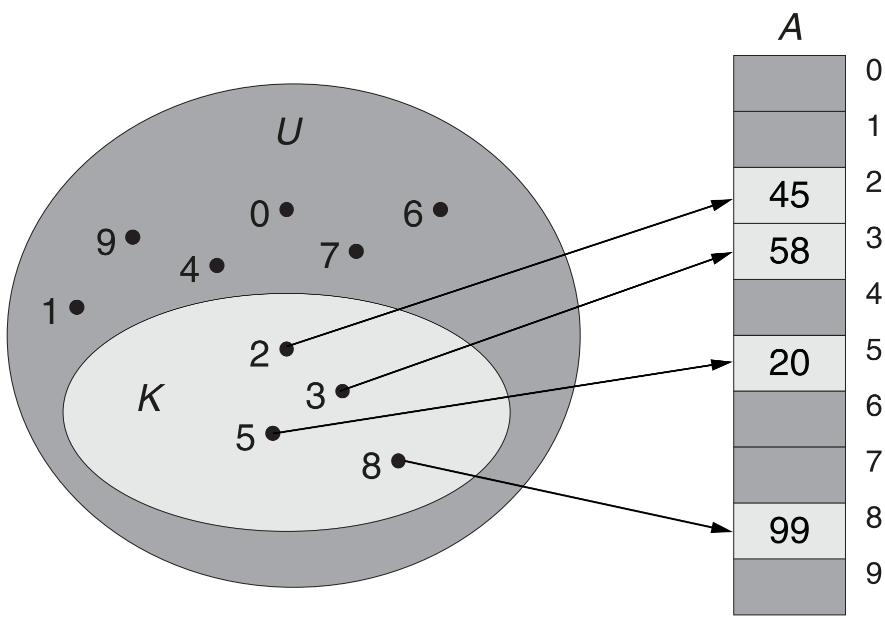
$K \subseteq U$ ist die Teilmenge tatsächlich benutzter Schlüssel

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Beispiel:



$U = \{0, 1, \ldots, m-1\}$ ist das Universum möglicher Schlüssel

A ist ein 0-indiziertes Array der Länge m

$K \subseteq U$ ist die Teilmenge tatsächlich benutzter Schlüssel

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Manipulation

Eingabe: A. Array der Länge m.
x. Einzufügendes / zu löschendes Element.
k. Schlüssel des fraglichen Elements.

Ausgabe: x. Gesuchtes Element.

Insert(A, x)

1. $A[x.\textit{key}] = x$

Search(A, k)

1. *return*$(A[k])$

Delete(A, x)

1. $A[x.\textit{key}] = \textit{NIL}$

Direct-address Table

Definition

Eine Direct-address Table (*direktadressierte Tabelle*) ist ein Dictionary, das für jeden möglichen Schlüsselwert genau einen Slot (Speicherplatz) eines Arrays vorhält.

Manipulation

Eingabe: A. Array der Länge m.
x. Einzufügendes / zu löschendes Element.
k. Schlüssel des fraglichen Elements.

Ausgabe: x. Gesuchtes Element.

Insert(A, x)

```
1.  A[x.key] = x
```

Search(A, k)

```
1.  return(A[k])
```

Delete(A, x)

```
1.  A[x.key] = NIL
```

Limitierungen

- Das Vorhalten von A mit $m = |U|$ Slots ist für große U unpraktisch / unmöglich.
- Bei $|K| \ll |U|$ wird der meiste für A benötigte Platz verschwendet.

Bemerkungen:

- Laufzeit: $\Theta(1)$ für alle Operationen im Worst-Case.
- Wenn der Datentyp der Satellitendaten der Records der abzuspeichernden Elemente es zulässt, können die Satellitendaten auch direkt im Array abgelegt werden. Das Speichern des Schlüssels als Teil des Elements ist dann redundant, da der Array-Index eindeutig anzeigt, welchen Schlüssel ein Element hat.

Hash Table

Definition

Eine Hash Table (*Hashtabelle*) ist ein Dictionary, das mit Hilfe einer Hashfunktion

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

das Universum U möglicher Schlüssel auf m Slots eines Arrays abbildet.

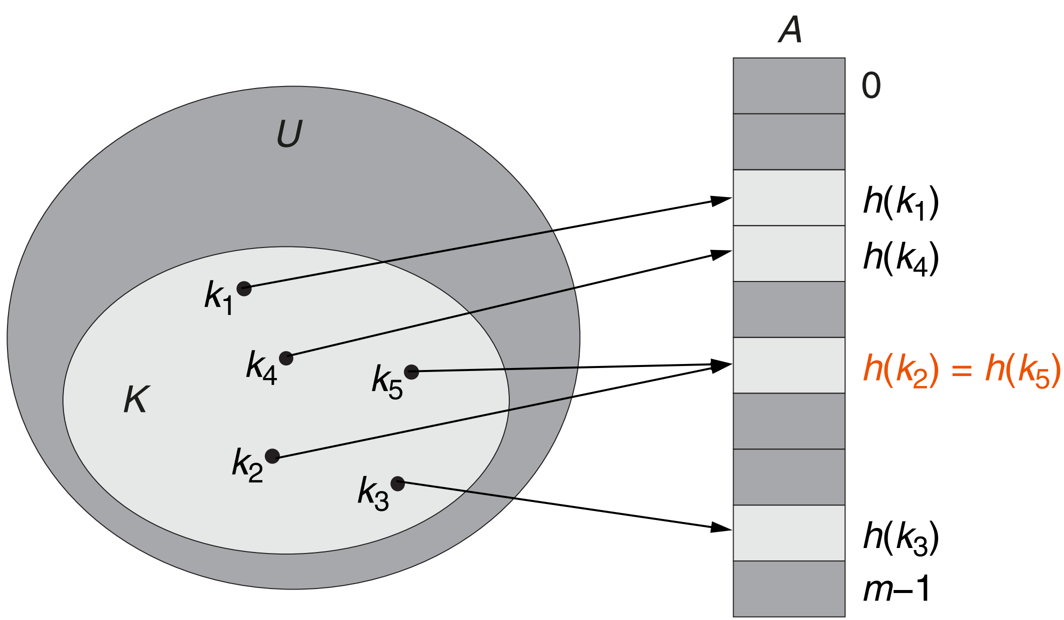
Hash Table
Definition

Eine Hash Table (*Hashtabelle*) ist ein Dictionary, das mit Hilfe einer Hashfunktion

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

das Universum U möglicher Schlüssel auf m Slots eines Arrays abbildet.

Beispiel:



Hash Table
Definition

Eine Hash Table (*Hashtabelle*) ist ein Dictionary, das mit Hilfe einer Hashfunktion

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

das Universum U möglicher Schlüssel auf m Slots eines Arrays abbildet.

Sei $K \subseteq U$ die Teilmenge tatsächlich benutzter Schlüssel. Der Fall, dass $h(k_i) = h(k_j)$ für Schlüssel $k_i, k_j \in K$ und $k_i \neq k_j$, heißt Hashkollision.

Dynamische Hash Tables implementieren eine Strategie zur Kollisionsbehandlung:

- Chaining (*Verkettung*)
  Verwendung von Linked Lists, um kollidierende Elemente im gleichen Slot abzulegen.

- Open Addressing (*offene Adressierung*) falls $|K| \leq m$
  Bestimmung eines alternativen, noch unbelegten Slots im Falle einer Kollision.

Statische Hash Tables implementieren eine perfekte Hashfunktion.

Bemerkungen:

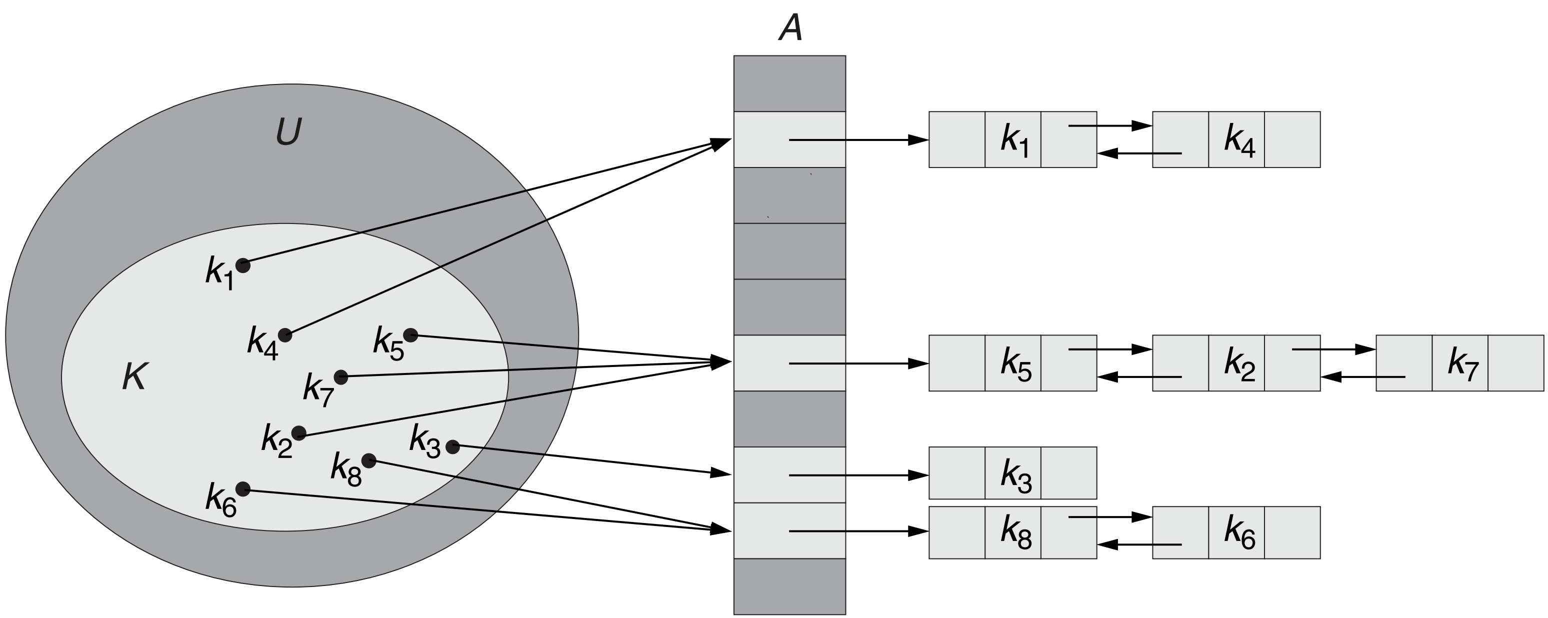
- Das Verb „to hash“ (*zerhacken*) weckt die passende Assoziation zufälligen Zusammenwürfelns von Schlüsselbestandteilen durch eine Hashfunktion.
- Das Universum von Schlüsseln kann, anders als bei der Direct-address Table, Schlüssel aller im Computer darstellbaren Datentypen beinhalten.
- Typischerweise sind $|U| > m$ und $|U| \gg |K|$. Andernfalls sind Direct-address Tables besser geeignet.

Hash Table

Chaining: Definition

Jeder Array-Slot enthält eine Variante einer Linked List, in die einzufügende Elemente eingereiht werden, um Hashkollisionen zu behandeln.

Beispiel:



Hash Table

Chaining: Manipulation

Eingabe: A. Array von m Linked Lists, in die n Elemente gemäß *HashInsert* und Hashfunktion h eingefügt wurden.
x. Einzufügendes / zu löschendes Element.
k. Schlüssel des gesuchten Elements.

Ausgabe: x. Gesuchtes Element.

HashInsert(A, x)

```
L = A[h(x.key)]
y = ListSearchKey(L, x.key)
IF y == NIL THEN
   ListInsertFront(L, x)
ENDIF
```

HashSearch(A, k)

```
L = A[h(k)]
return(ListSearchKey(L, k))
```

HashDelete(A, x)

```
L = A[h(x.key)]
ListDelete(L, x)
```

Hash Table

Chaining: Manipulation

Eingabe: A. Array von m Linked Lists, in die n Elemente gemäß *HashInsert* und Hashfunktion h eingefügt wurden.
x. Einzufügendes / zu löschendes Element.
k. Schlüssel des gesuchten Elements.

Ausgabe: x. Gesuchtes Element.

HashInsert(A, x)

$L = A[h(x.\mathit{key})]$
$y =$ *ListSearchKey*$(L, x.\mathit{key})$
IF $y == \mathit{NIL}$ **THEN**
    ListInsertFront(L, x)
ENDIF

HashSearch(A, k)

$L = A[h(k)]$
return(*ListSearchKey*(L, k))

HashDelete(A, x)

$L = A[h(x.\mathit{key})]$
ListDelete(L, x)

Hash Table

Chaining: Laufzeit

Annahme: Die Hashfunktion h lässt sich in $\Theta(1)$ Laufzeit errechnen.

Best Case:

- *HashInsert* und *HashSearch*: $T(n) = \Theta(1)$
  Entweder Liste L ist leer, oder x befindet sich am Anfang.

- *HashDelete*: $T(n) = \Theta(1)$
  Falls Doubly Linked Lists verwendet werden.

Worst Case:

- *HashInsert* und *HashSearch*: $T(n) = \Theta(n)$
  Liste L enthält alle Elemente und x befindet sich gegebenenfalls am Ende.

→ Voraussetzung: Die Hashfunktion h erzeugt ausschließlich Kollisionen.

- *HashDelete*: $T(n) = \Theta(1)$
  Falls Doubly Linked Lists verwendet werden.

Hash Table

Chaining: Laufzeit

Annahme: Simple Uniform Hashing (*Einfaches gleichverteiltes Hashing*)

- Die Wahrscheinlichkeit, mit der ein Schlüssel auf einen Slot gehasht wird, ist für jeden Slot $1/m$.

Load Factor (*Beladungsfaktor*):

- Unter der Annahme einfachen gleichverteilten Hashings, enthält eine Linked List enthält im Durchschnitt $\alpha = n/m$ Elemente.

Erwartete Listenlänge:

- Sei n_j die Länge der List $A[j]$ für $j \in [0, m-1]$.
- Es gilt $n = n_0 + n_1 + \ldots + n_{m-1}$.
- Die Erwartungswert $E[n_j] = \alpha = n/m$.

Hash Table

Satz 1 (Average-Case-Laufzeit I)

In einer Hash Table, in der Kollisionen mit Chaining behandelt werden, ist die Average-Case-Laufzeit unter der Annahme von Simple Uniform Hashing bei erfolgloser Suche in $\Theta(1 + \alpha)$.

Beweis:

Die Liste $A[h(k)]$ muss bis zum Ende betrachtet werden. Ihre erwartete Länge $E[n_{h(k)}] = \alpha$. Zuzüglich zur übrigen Laufzeit beträgt die Gesamtlaufzeit $\Theta(1 + \alpha)$. $\square$

Hash Table

Satz 1 (Average-Case-Laufzeit I)

In einer Hash Table, in der Kollisionen mit Chaining behandelt werden, ist die Average-Case-Laufzeit unter der Annahme von Simple Uniform Hashing bei erfolgloser Suche in $\Theta(1+\alpha)$.

Beweis:

Die Liste $A[h(k)]$ muss bis zum Ende betrachtet werden. Ihre erwartete Länge $E[n_{h(k)}] = \alpha$. Zuzüglich zur übrigen Laufzeit beträgt die Gesamtlaufzeit $\Theta(1+\alpha)$. □

Satz 2 (Average-Case-Laufzeit II)

In einer Hash Table, in der Kollisionen mit Chaining behandelt werden, ist die Average-Case-Laufzeit unter der Annahme von Simple Uniform Hashing bei erfolgreicher Suche in $\Theta(1+\alpha)$.

Beweisidee:

Probabilistische Analyse der erwarteten Anzahl von Elementen, die untersucht werden müssen, um x zu finden, nämlich die Elemente, die nach x eingefügt wurden. Die sind in $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$.

Bemerkungen:

- Ein Load Factor von $\alpha < 1$ bedeutet, dass weniger Elemente als Slots gespeichert werden müssen, $\alpha = 1$ bedeutet, dass die Zahl der Elemente und Slots gleich ist, und $\alpha > 1$, dass mehr Elemente als Slots gespeichert werden müsse, so dass Kollisionen unvermeidlich sind.
- Selbst bei einem Load Factor von $\alpha < 1$ sind Kollisionen unter der Annahme von Simple Uniform Hashing schon bei wenigen zu speichernden Elementen wahrscheinlich. Zum Verständnis kann das sogenannte Geburtstagsparadoxon herangezogen werden: „Befinden sich in einem Raum mindestens 23 Personen, dann ist die Chance, dass zwei oder mehr dieser Personen am gleichen Tag (ohne Beachtung des Jahrganges) Geburtstag haben, größer als 50%."
- Die Fallunterscheidung ist notwendig, da im Falle einer erfolgreichen Suche die Wahrscheinlichkeit, mit der eine Liste nach x durchsucht wird, proportional zu ihrer Länge ist, was bei erfolgloser Suche nicht der Fall ist.

Hash Table

Chaining: Laufzeit

Interpretation:

- Wähle die Zahl der Slots m proportional zur erwarteten Zahl der Elemente n.
- Dann gilt $n = O(m)$, so dass

$$\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1).$$

➜ Eine Hash Table kann im Schnitt in $O(1)$ Laufzeit manipuliert werden.

Hash Table

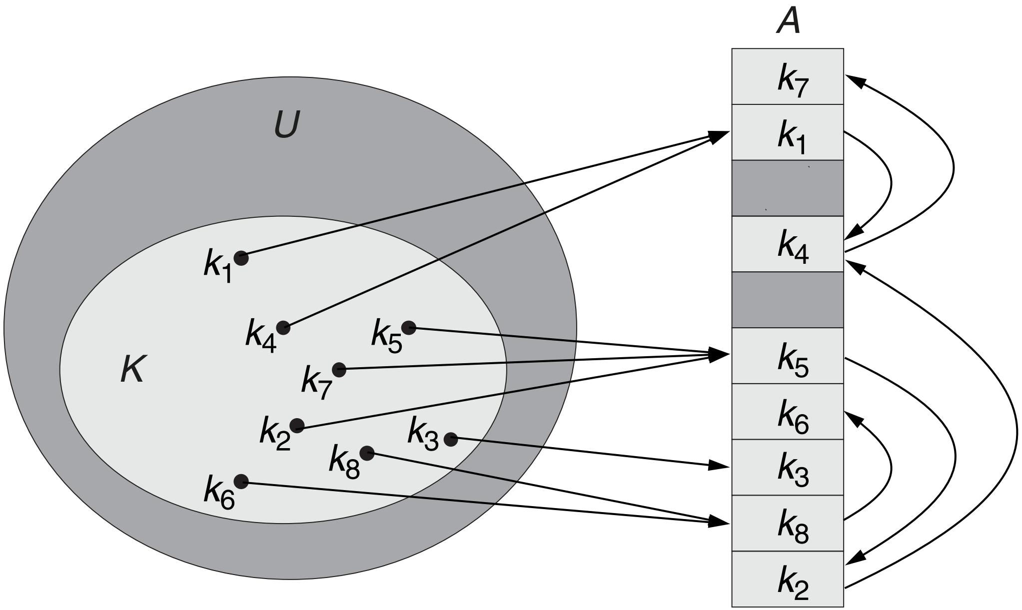
Open Addressing: Definition

Jeder Array-Slot enthält maximal ein Element. Es werden mittels Hashfunktion

$$h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Sondierversuch}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Slot-Index}}$$

iterativ alle Array-Slots sondiert, bis ein freier Slot gefunden wurde.

Beispiel:



$h(k_7, 3)$

$h(k_1, 0) = h(k_4, 0)$

$h(k_4, 1) = h(k_7, 2)$

$h(k_5, 0) = h(k_2, 0) = h(k_7, 0)$

$h(k_6, 1)$

$h(k_3, 0)$

$h(k_8, 0) = h(k_6, 0)$

$h(k_2, 1) = h(k_7, 1)$

Hash Table

Open Addressing: Definition

Jeder Array-Slot enthält maximal ein Element. Es werden mittels Hashfunktion

$$h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Sondierversuch}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{Slot-Index}}$$

iterativ alle Array-Slots sondiert, bis ein freier Slot gefunden wurde.

Die Folge von Slot-Indexen, die für ein gegebenes k durch $h(k, i)$ für $i = 0$ bis $i = m - 1$ berechnet wird muss eine Permutation von $\{0, 1, \ldots, m - 1\}$ sein.

Heuristiken zum Probing (*Sondieren*):

- Linear Probing (*Lineares Sondieren*)
  Bestimmung des freien Slots mit dem nächstgrößten Index.
- Quadratic Probing (*Quadratisches Sondieren*)
  Bestimmung des freien Slots mit Hilfe einer quadratischen Funktion.
- Double Hashing (*Doppel-Hashing*)
  Bestimmung des freien Slots mit Hilfe zweier Hashfunktionen.

Hash Table

Open Addressing: Manipulation

Eingabe: A. Array von m Slots, in das n Elemente gemäß *HashInsert* und Hashfunktion h eingefügt wurden.
k. Schlüssel des einzufügenden / gesuchten / zu löschenden Elements.

Ausgabe: k. Schlüssel des eingefügten / gesuchten Elements.

HashInsert(A, k)

$i = 0$
DO
  $j = h(k, i)$
  IF $A[j] ==$ *NIL*
  OR $A[j] ==$ *DEL* **THEN**
    $A[j] = k$
    return(j)
  ELSE
    $i = i + 1$
  ENDIF
WHILE $i == m$;
error(“overflow”)

HashSearch(A, k)

$i = 0$
DO
  $j = h(k, i)$
  IF $A[j] == k$ **THEN**
    return(j)
  ENDIF
  $i = i + 1$
WHILE $A[j] ==$ *NIL*
  OR $A[j] ==$ *DEL*
  OR $i == m$;
return(*NIL*)

HashDelete(A, k)

$j =$ *HashSearch*(A, k)
IF $A[j] \neq$ *NIL* **THEN**
  $A[j] =$ *DEL*
ENDIF

Bemerkungen:

- Beim Löschen eines Elements genügt es nicht, den entsprechenden Slot auf *NIL* zu setzen, sondern es wird ein spezielles Symbol *DEL* benötigt. Andernfalls würden Element unauffindbar, die mit einem vorher eingefügten und dann gelöschten Elementen kollidieren. Beim Einfügen und bei der Suche muss daher das Symbol *DEL* zusätzlich zu *NIL* berücksichtigt werden.

  Nachteil dieses Vorgehens ist, dass die Laufzeit für die Suche nach Elementen nun nicht mehr auf dem tatsächlichen Load Factor α der Hash Table beruht, da nach einer langen Folge von Einfügungen und Löschungen mit der Zeit alle Array-Slots entweder belegt oder als gelöscht markiert sind.

- Eine Lösung besteht darin, den Load Factor inklusive der gelöschten Elemente zu berechnen und wenn der Load Factor sich 1 annähert, die Hash Table (mit gegebenenfalls größerem Array) und ohne die gelöschten Elemente neu aufzubauen.

Hash Table

Open Addressing: Probing-Heuristiken

Annahme: Uniform Hashing (*Gleichverteiltes Hashing*)

- Die Wahrscheinlichkeit, mit der ein Schlüssel eine der $m!$ möglichen Permutationen von $\{0, 1, \ldots, m-1\}$ als Sondierungssequenz hat ist $1/m!$.
- Hashfunktionen, die die Uniform-Hashing-Annahme erfüllen, sind schwer zu implementieren.
- In der Praxis werden heuristische Annäherungen eingesetzt, die weniger Sondierungssequenzen erzeugen können.

Hilfshashfunktion h':

- Die Heuristiken verwenden herkömmliche Hilfshashfunktionen h' der Form

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

- Für h' gilt Simple Uniform Hashing.

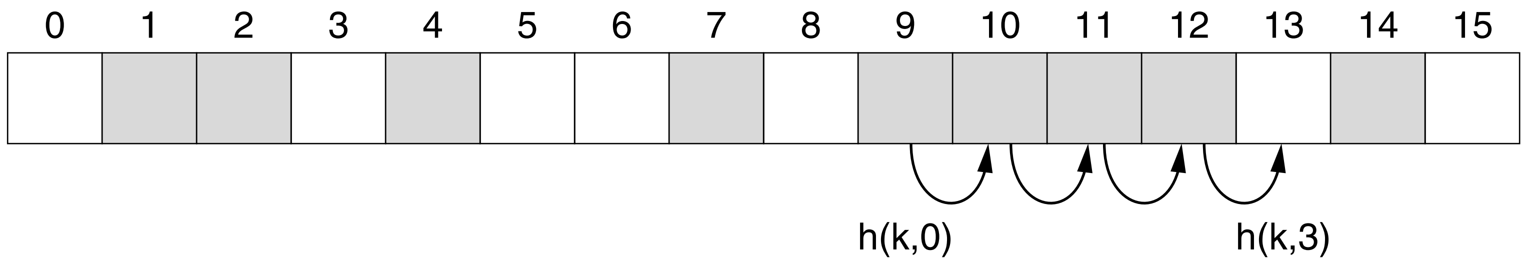
Hash Table

Open Addressing: Linear Probing

Hashfunktion:

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

Beispiel:



Eigenschaften:

- h induziert m verschiedene Sondierungssequenzen.
- Für alle $U \times \{0, 1, \ldots, m-1\}$ sind alle m Slot-Indexe in der Bildmenge von h.

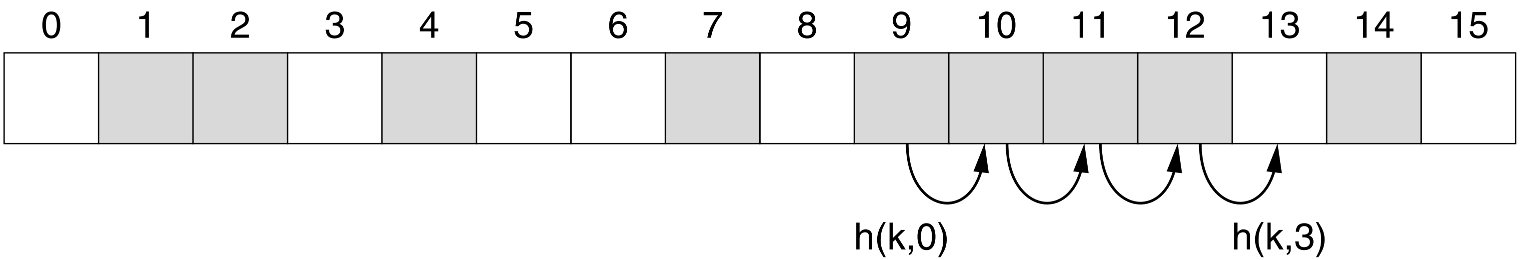
Hash Table

Open Addressing: Linear Probing

Hashfunktion:

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

Beispiel:



Eigenschaften: (Fortsetzung)

- Primary Clustering (*Primäre „Gruppenbildung“*)
  Es bilden sich Ketten belegter Slots. Je länger eine Kette, desto wahrscheinlicher wird der nachfolgende Slot belegt. Ketten werden vereinigt, so dass mit steigendem Load Factor die Suchzeit in $O(n)$ ist.

- Secondary Clustering (*Sekundäre „Gruppenbildung“*)
  Kollidierende Schlüssel erzeugen dieselbe Sondierungssequenz.

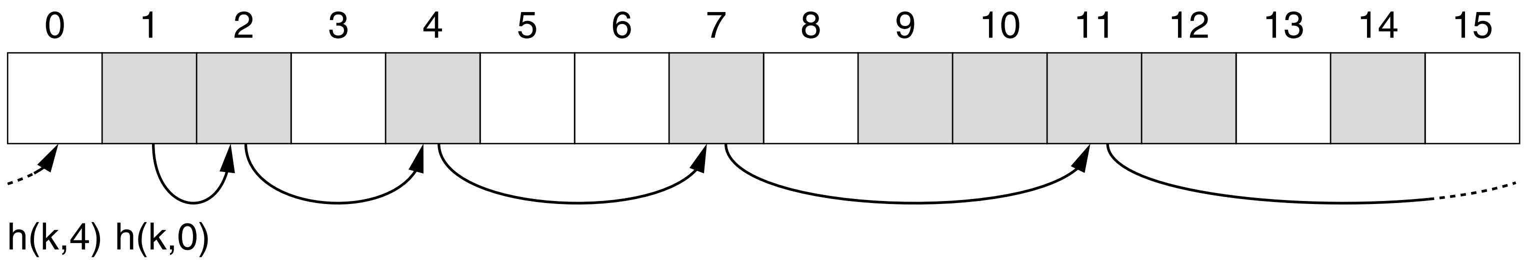
Hash Table

Open Addressing: Quadratic Probing

Hashfunktion mit Konstanten c_1 und $c_2 \neq 0$:

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

Beispiel:



Eigenschaften:

- h induziert m verschiedene Sondierungssequenzen.
- Für $m = 2^n$ und $c_1 = c_2 = 1/2$ sind alle m Slot-Indexe in der Bildmenge von h.
- Für Primzahl $m > 2$, $c_1 = c_2 = 1/2$, $c_1 = c_2 = 1$ oder $c_1 = 0$, $c_2 = 1$, und $\alpha < 0.5$ wird immer ein leerer Slot gefunden und keine Slots werden doppelt geprüft.
- Secondary Clustering

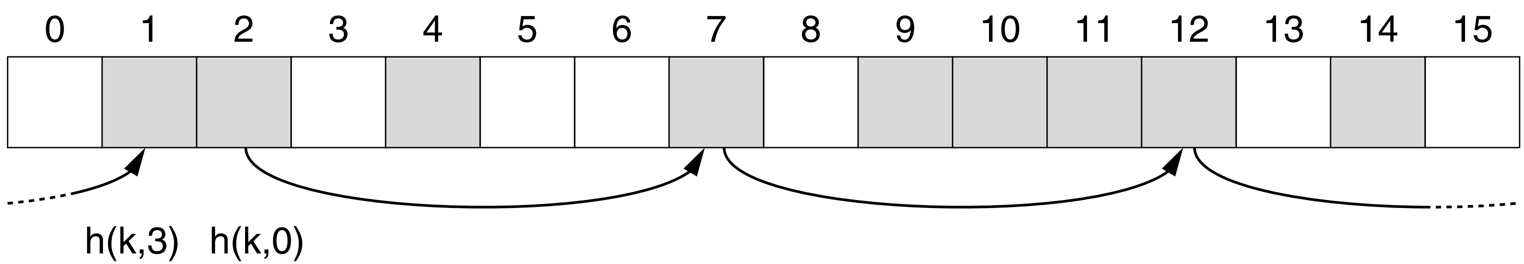
Hash Table

Open Addressing: Double Hashing

Hashfunktion mit zwei Hilfshashfunktionen h'_1 und h'_2:

$$h(k, i) = (h'_1(k) + i \cdot h'_2(k)) \bmod m$$

Beispiel:



Eigenschaften:

- Ist $h'_2(k)$ teilerfremd zu m sind alle m Slot-Indexe in der Bildmenge von h.
  Dies ist zum Beispiel gewährleistet, wenn m eine Zweierpotenz ist und h'_2 ungerade Zahlen berechnet oder wenn m prim ist und h_2 eine positive Zahl kleiner m berechnet.
- Das Sprungintervall wird durch h'_2 abhängig vom Schlüssel k erzeugt.
- Unter den vorigen Bedingungen werden m^2 Sondierungssequenzen von h induziert.

Hash Table

Open Addressing: Laufzeit

Abhängigkeiten:

- Alle Manipulationsalgorithmen Suchen nach einem Schlüssel k.
- Die Suchzeit hängt von der Länge der Sondierungssequenz ab.
- Die Länge der Sondierungssequenz hängt vom Load Factor $\alpha = n/m$ ab.

Worst Case und Best Case sind analog zum Chaining.

Average Case:

- Entspricht der erwarteten Länge der Sondierungssequenz.
- Idealisierte Sicht: Uniform Hashing
- Praktische Sicht: Heuristiken

Hash Table

Open Addressing: Laufzeit

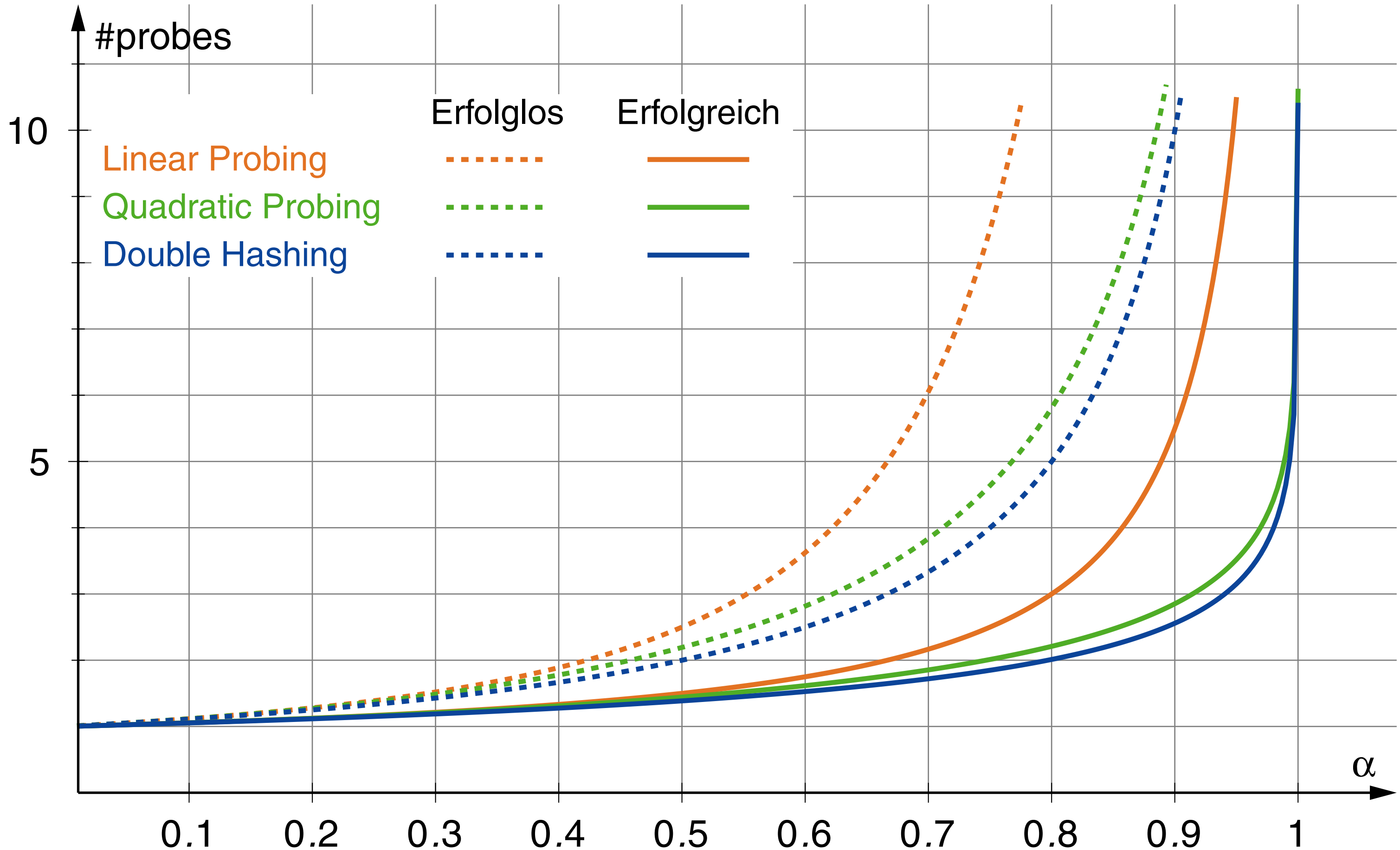
Erwartete Länge einer Sondierungssequenz:

Probing	Suche erfolglos	Suche erfolgreich
Uniform Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$
Linear Probing	$\frac{1}{2}\left(1+\frac{1}{(1+\alpha)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{1+\alpha}\right)$
Quadratic Probing	$\frac{1}{1-\alpha}+\ln\left(\frac{1}{1-\alpha}\right)-\alpha$	$1+\ln\left(\frac{1}{1-\alpha}\right)-\frac{\alpha}{2}$
Double Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

Hash Table

Open Addressing: Laufzeit

Erwartete Länge einer Sondierungssequenz:



Hash Table

Open Addressing: Laufzeit

Erwartete Länge einer Sondierungssequenz:

Load	**Suche erfolglos**			**Suche erfolgreich**		
Factor	Linear	Quadratic	Double	Linear	Quadratic	Double
$\alpha = 0.5$	2.5	2.19	2	1.5	1.44	1.39
$\alpha = 0.75$	8.5	4.64	4	2.5	2.01	1.84
$\alpha = 0.9$	50.5	11.40	10	5.5	2.85	2.55
$\alpha = 0.95$	200.5	22.05	20	10.5	3.52	3.15