Teil 1: Suchen

- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
  - Idee
  - Hashfunktion
  - Hashverfahren mit Verkettung
  - Offene Hashverfahren
  - Dynamische Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume
- Heaps

Sequentielle Suche

Laufzeit (Worst-Case):

Operation	T(n)
search *)	O(n)
insert *)	O(n) **)
remove *)	O(n)
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	O(n^2)
n search-Aufrufe *)	O(n^2)

*) bei einer Menge mit n Elementen.

**) Es muss geprüft werden, ob das Element bereits vorkommt (search-Aufruf)

Binäre Suche

Laufzeit (Worst-Case):

Operation	T(n)
search [1)]	$O(\log n)$
insert [1)]	$O(n)$
remove [1)]	$O(n)$
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	$O(n^2)$ [2)]
n search-Aufrufe [1)]	$O(n \log n)$

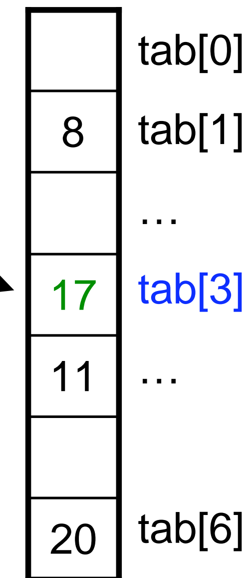
[1)] Bei einer Menge mit n Elementen.

[2)] Alternativ lassen sich auch n Elemente unsortiert einfügen und dann mit einem schnellen Sortierverfahren sortieren.
Damit würde man $T(n) = O(n \log n)$ erreichen.

Idee der Hashverfahren

- Mit einer Hashfunktion h wird aus dem Schlüssel k eine Hashadresse h(k) (positive ganze Zahl) berechnet.
- Die Hashadresse gibt den Index in einem Feld an, wo der Datensatz abgespeichert werden kann bzw. abgespeichert ist.
  Das Feld wird auch Hashtabelle genannt.

Schlüssel z.B. k = 17 → Hashfunktion h → Hashadresse z.B. h(k) = 3



Hashtabelle tab

Damit lassen sich die Dictionary-Operationen sehr einfach realisieren:

- search(k): return tab[h(k)];
- insert(k, v): tab[h(k)] = v;
- remove(k): tab[h(k)] = „empty";

Im Idealfall alle Operationen in O(1)!

Hashfunktion (1)

Wichtige Anforderung an eine Hashfunktion

- Sollte einfach zu berechnen sein.
- Gute Streuwirkung:
  vorkommende Schlüssel sollten sich möglichst gut über die Tabelle verteilen.

Wichtige Hashfunktionen

- **Division-Rest-Methode:**

  h(k) = k mod m;

  Hierbei ist m die Tabellengröße. m sollte möglichst eine Primzahl sein.

- **Multiplikative Methode:**

  $h(k) = \lfloor m(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor) \rfloor$

  Dabei ist $\phi^{-1} = (\sqrt{5} - 1)/2 \approx 0.6180339887$ der Kehrwert des goldenen Schnitts.
  $\lfloor x \rfloor$ rundet auf die nächstkleinere ganze Zahl ab und m ist die Tabellengröße.

  Beispielsweise bilden die Werte h(1), h(2), …, h(10) für m = 10 eine Permutation der Zahlen 0,1, …, 9 nämlich: 6, 2, 8, 4, 0, 7, 3, 9, 5, 1.

Hashfunktion (2)

Beispiel:

Füge die Zahlen

7, 24, 5, 8

in eine Hashtabelle der Größe m = 7 ein.

Benutze die Hashfunktion

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

Wert	Index
7	tab[0]
8	tab[1]
	tab[2]
24	tab[3]
	tab[4]
5	tab[5]
	tab[6]

Hashfunktion (3)

Wahl einer Hashfunktion bei String-Schlüssel

Stellenwertmethode:

Fasse String als Zahl im Stellenwertsystem zur Basis b = 128 (Anzahl ASCII-Zeichen) auf und rechne String in Dezimalzahl um. Wende dann die Divisionsrest-Methode an.

Beispiel:

$$h(\text{"TEST"}) = (\,\text{ascii('T')}*128^3 + \text{ascii('E')}*128^2 + \text{ascii('S')}\ 128^1 + \text{ascii('T')}*128^0\,) \bmod m$$
$$= (84*128^3 + 69*128^2 + 83*128^1 + 84*128^0) \bmod m$$

ASCII-Wert von 'T'

Um die Gefahr eines Überlaufs bei langen Strings zu vermeiden, wende das Horner-Schema an und ziehe die mod-Operation in die Klammern:

$$h(\text{"TEST"}) = [\,84*128^3 + 69*128^2 + 83*128^1 + 84*128^0\,] \bmod m$$
$$= [\,((84*128 + 69)*128 + 83)*128 + 84\,] \bmod m$$
$$= [\,((84 \bmod m *128 + 69) \bmod m *128 + 83) \bmod m *128 + 84\,] \bmod m$$

C++-Funktion:

```
int h(const string& k, int m) {
	int adr = 0;
	for (int i = 0; i < k.size(); i++)
		adr = (adr*128 + k[i]) % m;
	return adr;
}
```

Hashfunktion (4)

Adresskollision

Im allgemeinen ist eine Hashfunktion nicht injektiv.
D.h. unterschiedliche Schlüssel können die gleiche Hashadresse haben.

Beispiel:

Mit $h(k) = k \bmod m$ mit $m = 7$ gilt:

$h(11) = h(25) = h(74) = 4$.

Geburtstagsparadoxon

In einer Gruppe von 23 Personen ist es wahrscheinlich (nämlich $P = 0.51$), dass 2 Personen am gleichen Tag Geburtstag haben.

Konsequenz: würde man die Daten von 23 Personen in eine Tabelle der Größe 365 mit einem Hashverfahren abbilden und als Hashfunktion

$h(p)$ = Geburtstag der Person p auf $\{0, 1, 2, \ldots, 364\}$ umgerechnet

wählen, dann würde es wahrscheinlich eine Kollision geben.

Kollisionsbehandlung:

- Hashverfahren mit Verkettung
- Offene Hashverfahren

Hashfunktion (5)

Aufgabe 1.1

Beurteilen Sie folgende Hashfunktionen:

a) Die Personaldaten einer Firma sollen in einer Hashtabelle abgespeichert werden. Dazu soll die Personalnummer als Schlüssel verwendet werden. Die letzte Stelle der Personalnummer gibt das Geschlecht an: 0 = männlich und 1 = weiblich. Als Hashfunktion werde $h = k \bmod m$ mit einer geraden Tabellengröße m gewählt.

b) Die Personalnummer wird nun 6-stellig gewählt, wobei die vordersten 3 Stellen die Abteilung codieren, in der die Person arbeitet. Als Hashfunktion werde $h = k \bmod m$ mit einer Tabellengröße $m = 1000$ gewählt.

c) Ein Telefonbuch mit etwa 100000 Einträgen soll als Hashtabelle realisiert werden. Die Hashfunktion muss Familiennamen (Strings) in Hashadressen umrechnen. Als Hashfunktion wird folgendes gewählt:

$$h(\text{"}z_0 z_1 \ldots z_{n-1}\text{"}) = [\, \mathrm{ascii}(z_0) + \mathrm{ascii}(z_1) + \ldots + \mathrm{ascii}(z_{n-1}) \,] \bmod m,$$

wobei die Tabellengröße m irgendeine Zahl größer als 100000 ist.

Aufgabe 1.2

Geben sie eine geeignete Hashfunktion für Kalenderdaten der Form tt.mm.jjjj an.

Einschub: Verteilung der Primzahlen

Primzahlfunktion

$\pi(n)$ = Anzahl der Primzahlen $\leq$ n.

Primzahlsatz

$\pi(n) \sim n/\ln(n)$

Damit gilt für die Primzahlhäufigkeit:

$\pi(n)/n \sim 1/\ln(n)$

Tabelle zeigt die tatsächliche und die geschätzte Primzahlhäufigkeit

n	$\pi(n)/n$	$1/\ln(n)$
10^4	0.123	0.11
10^5	0.096	0.087
10^6	0.078	0.072
10^7	0.066	0.062
10^8	0.058	0.054
10^9	0.051	0.048
10^{100}	?	0.0043
10^{200}	?	0.0021

Im praktisch relevanten Bereich liegt die Primzahlhäufigkeit bei 5 bis 10%.

Hashverfahren mit Verkettung (1)

Idee:

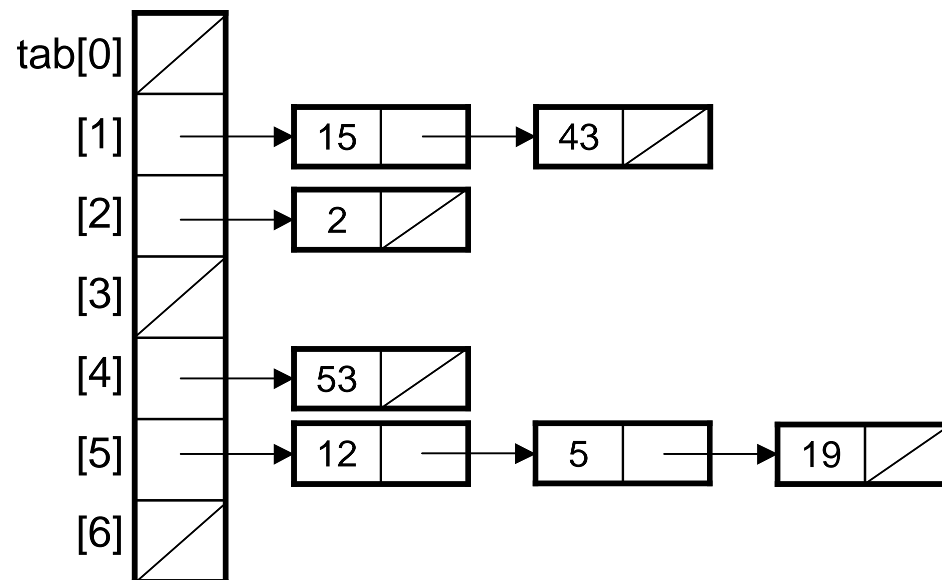
Der Hashtabellen-Eintrag tab[i] zeigt auf eine **verkette Liste**, die alle Datensätze mit der gleichen Hashadresse i enthalten.

Beispiel

Für m = 7 mit h(k) = k mod m ergibt sich nach dem Einfügen von

12, 53, 5, 15, 2, 19, 43

folgende Hashtabelle:



Hashverfahren mit Verkettung (2)

Algorithmen:

```
bool search (Key k)
{
    suche in tab[ h(k) ] nach Schlüssel k;
    if (k gefunden)
        return true;
    else
        return false;
}
```

```
bool insert (Key k)
{
    suche in tab[ h(k) ] nach Schlüssel k;
    if (k gefunden)
        // Schlüssel bereits vorhanden:
        return false;
    else {
        füge k am Ende oder
        am Anfang der Liste an;
        return true;
    }
}
```

```
bool remove (Key k)
{
    suche in tab[ h(k) ] nach Schlüssel k;
    if (k gefunden) {
        entferne Knoten k aus Liste;
        return true;
    }
    else
        return false;
}
```

Offene Hashverfahren (1)

Idee:

- Alle Datensätze werden in einem Feld (d.h. keine verkettete Listen) untergebracht.
- Falls beim Eintragen des Schlüssels k die Adresse h(k) bereits belegt ist, wird gemäß einer **Sondierungsfolge** die erste **freie (offene) Adresse** gesucht und k dort abgespeichert.
- Beim Suchen von k wird ebenfalls die Sondierungsfolge durchlaufen.
- In der engl. Literatur auch als **open hashing** bezeichnet.

Allgemeine Sondierungsfolge

$h(k) + s(j,k) \bmod m$ mit $j = 0, 1, \ldots, m-1$

Beispiel für eine Sondierungsfolge

$s(j,k) = j$

Damit ergibt sich die Sondierungsfolge:

$h(k) + 0 = h(k)$
$h(k) + 1 \bmod m$
$h(k) + 2 \bmod m$
…

Offene Hashverfahren (2)

Beispiel

Für m = 7 mit

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

der Sondierungsfolge

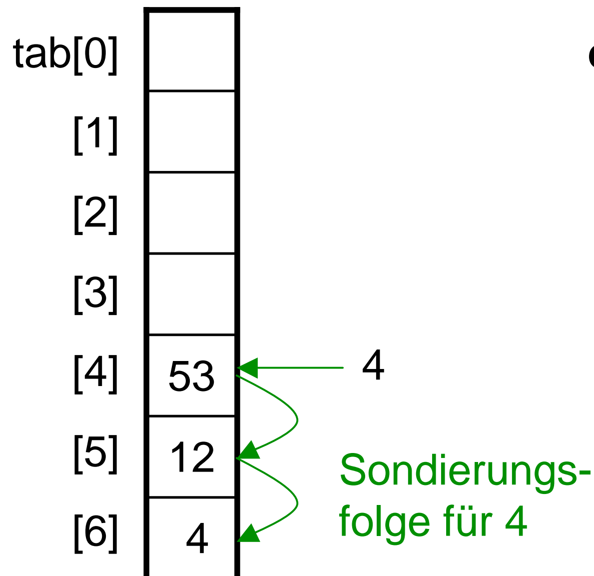
$s(j,k) = j$

ergibt sich nach dem Einfügen von

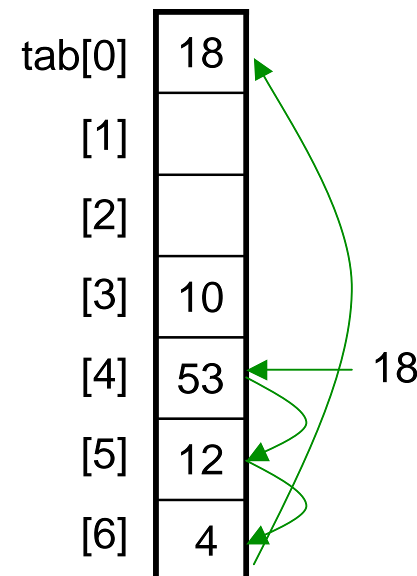
12, 53, 4, 10, 18

folgende Hashtabelle:

12, 53 und 4 eingefügt:



10 und 18 eingefügt:



Offene Hashverfahren (3)

Algorithmus zum Suchen:

```
bool search (Key k, Value& v)
{
    int j = 0;
    do {
        adr = (h(k) + s(j,k)) % m;
        j++;
    } while (tab[adr].key != ″leer″ && tab[adr].key != k);

    if (tab[adr].key != ″leer″) {
        v = tab[adr].value;
        return true;
    }
    else
        return false;
}
```

Offene Hashverfahren (4)

Algorithmus zum Löschen:

```
bool remove (Key k)
{
	int j = 0;
	do {
		adr = (h(k) + s(j,k)) % m;
		j++;
	} while (tab[adr].key != ″leer″ && tab[adr].key != k);

	if (tab[adr].key != ″leer″) {
		tab[adr].key = ″gelöscht″ ;
		return true;
	}
	else
		return false;
}
```

Wichtig:

Eine Hashadresse hat 3 Zustände:

- Eintrag vorhanden
- Eintrag gelöscht
- Eintrag leer

Zu Beginn sind alle Einträge leer.

Grund:
Beim Löschen können Lücken entstehen, die bei einer späteren Suchoperation übersprungen werden müssen.

Offene Hashverfahren (5)

Algorithmus zum Einfügen:

```
bool insert (Key k, Value v)
{
    if (search(k))   // k ist bereits vorhanden
        return false;

    int j = 0;
    do {
        adr = (h(k) + s(j,k)) % m;
        j++;
    } while (tab[adr].key != "leer" || tab[adr].key  != "gelöscht");

    tab[adr].key = k;
    tab[adr].value = v;
    return true;
}
```

Es werden zuerst wieder die Lücken gefüllt

Wichtig:

Um Endlosschleifen bei stark gefüllten Tabellen zu vermeiden, sind die Längen der Sondierungsfolgen zu beschränken.

Sondierungsarten (1)

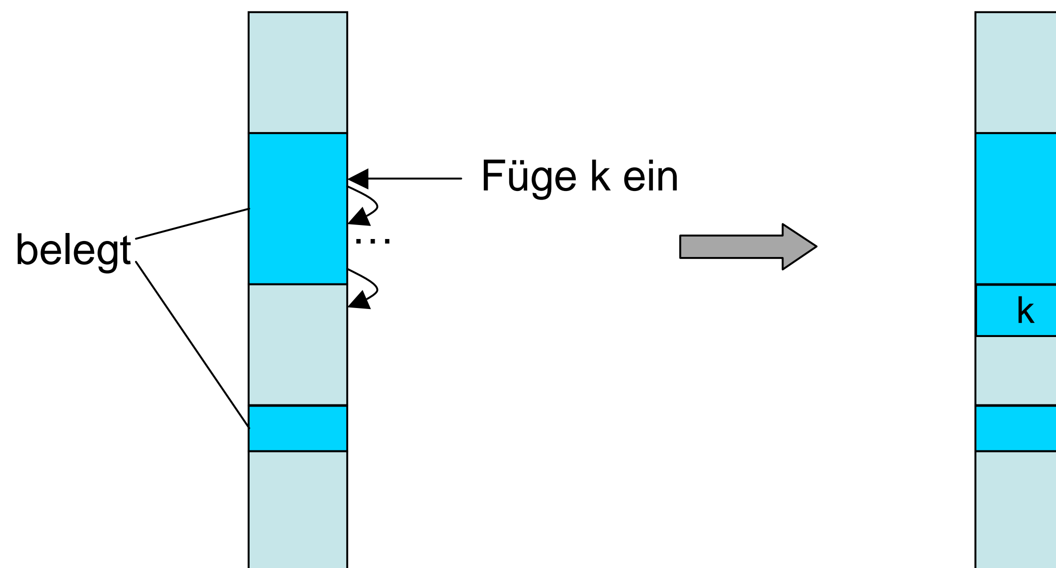
Lineares Sondieren (linear probing)

$s(j,k) = j$

Damit ergibt sich die Sondierungsfolge:

$h(k)$
$h(k) + 1 \bmod m$
$h(k) + 2 \bmod m$
…

Lineares Sondieren tendiert aufgrund von Sekundärkollisionen (zwei Sondierungsfolgen überschneiden sich) zu Clusterbildung:
Große belegte Cluster haben eine stärkere Tendenz zu wachsen als kleinere.



Sondierungsarten (2)

Quadratisches Sondieren (quadratic probing)

$s(j,k) = j^2$

Damit ergibt sich die Sondierungsfolge:

h(k)
h(k) + 1 mod m
h(k) + 4 mod m
h(k) + 9 mod m
…

Quadratisches Sondieren streut wesentlich besser als lineares Sondieren.

Achtung:

- es gibt keine Garantie dafür, daß sich bei einer genügend gefüllten Tabelle immer ein freier Platz findet.
- wenn die Tabellengröße eine Primzahl ist, dann findet sich in einer weniger als halbvollen Tabelle immer ein Platz.

Sondierungsarten (3)

Problem:

Es wird mit einer quadratischen Sondierungsfolge im allgemeinen nicht jede Adresse erreicht.

Beispiel:

Für m = 8 ergibt sich mit h(k) = 0 folgende Sondierungsfolge:

0, 0 + 1 mod m = 1,
0 + 4 mod m = 4, 0 + 9 mod m = 1,
0 + 16 mod m = 0, 0+ 25 mod m = 1, …

Eigenschaften:

- Wenn m eine Primzahl ist, dann wird mit jeder Sondierungsfolge wenigstens die Hälfte aller Einträge erreicht.
- Ist m eine Primzahl der Form 4i + 3 und wird die alternierende Sondierungsfolge

  $$s(j,k) = \lceil j/2 \rceil^2 (-1)^j$$

  gewählt, dann werden alle Einträge erreicht.
  Die Sondierungsfolge lautet konkret:

  h(k), h(k) + 1, h(k) – 1, h(k) + 4, h(k) – 4, …

Sondierungsarten (4)

Double Hashing

s(j,k) = j * h'(k)

Dabei ist h' eine weitere Hashfunktion.

Damit ergibt sich folgende Sondierungsfolge:

h(k)
h(k) + h'(k) mod m
h(k) + 2*h'(k) mod m
h(k) + 3*h'(k) mod m
…

Die beiden Hashfunktionen sollten möglichst unabhängig sein.

Eine gute Wahl ist:

h(k) = k mod m und
h'(k) = (k+1) mod (m-2).

Außerdem ist es wichtig, dass die Tabellengröße m eine Primzahl ist.

Nur dann erreicht jede Sondierungsfolge alle Einträge.

Sondierungsarten (5)

Aufgabe 1.3

Führen Sie mit offenem Hashing und quadratischem Sondieren mit der Sondierungsfolge $s(j,k) = j^2$ für eine Tabelle der Größe $m = 7$ folgende Operationen durch:

- einfügen von 3, 4, 10;
- löschen von 4;
- suchen von 10 und 25
- einfügen von 17.

Führen Sie mit der alternierenden Sondierungsfolge $s(j,k) = \lceil j/2 \rceil^2 (-1)^j$ folgende Operationen durch:

- einfügen von 3, 4, 10, 9.

Analyse der Hashverfahren (1)

Analyse im schlechtesten Fall

Alle n Einträge haben die gleiche Hashadresse.
Daher muss jede Operation eine Liste mit bis zu n Einträgen ablaufen.

Dies ist in der Praxis so unwahrscheinlich, dass hier die Analyse im mittleren Fall wesentlich wichtiger ist.

Analyse im mittleren Fall

Wichtige Maßzahlen:

C   Anzahl der durchschnittlich betrachteten Einträge bei erfolgreicher Suche.

C‘   Anzahl der durchschnittlich betrachteten Einträge bei nicht erfolgreicher Suche.

Es zeigt sich, dass C und C‘ nur abhängig sind vom Belegungsfaktor:

$\alpha = n/m$;   n = Anzahl der Einträge und m = Tabellengröße.

Analyse der Hashverfahren (2)

C und C‘ für verschiedene Hashverfahren

Verfahren	C	C’
Hashverfahren mit Verkettung	$1+\frac{\alpha}{2}$	α
Offenes Hashverfahren mit linearem Sondieren	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
Offenes Hashverfahren mit quadratischem Sondieren	$1+\ln\left(\frac{1}{1-\alpha}\right)-\frac{\alpha}{2}$	$\frac{1}{1-\alpha}-\alpha+\ln\left(\frac{1}{1-\alpha}\right)$
double hashing mit unabhängigen h u. h’	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$

Die Angaben für C und C‘ setzen eine ideale Hashfunktion voraus.
D.h. die Schlüssel sind gleichmäßig über die Tabelle verstreut.

Die umfangreichen Herleitungen können in [Ottmann u. Widmayer 2002] nachgelesen werden.

Analyse der Hashverfahren (3)

C und C' für konkrete Belegungsfaktoren:

	α = 0.5		α = 2/3		α = 0.8	
Verfahren	**C**	**C'**	**C**	**C'**	**C**	**C'**
Hashverfahren mit Verkettung	1.25	0.5	1.33	0.66	1.4	0.8
Offenes Hashverfahren mit linearem Sondieren	1.5	2.5	2	5	3	13
Offenes Hashverfahren mit quadratischem Sondieren	1.44	2.19	1.77	3.43	2.21	5.81
double hashing mit unabhängigen h u. h'	1.38	2	1.65	3	2.01	5

Bei offenen Hashverfahren wird in der Praxis üblicherweise ein Belegungsfaktor von

$\alpha \leq 2/3$

angestrebt.

Dynamische Hashverfahren (1)

Problem

Bei den bisher besprochenen Hashverfahren wird die Tabellengröße m einmalig festgelegt und die Hashfunktion damit parametrisiert.

Wird ein bestimmter Füllungsgrad überschritten, dann kann das eingesetzte Hashverfahren sehr langsam werden, so dass eine Umorganisation notwendig wird.

Vergrößern der Tabelle mit sofortigem Umkopieren:

Falls beim Einfügen ein bestimmter Füllungsgrad überschritten wird, werden folgende Operationen durchgeführt:

- Lege neue Tabelle tab‘ mit einer in etwa doppelten Größe m‘ an;
- Kopiere alle Einträge aus alter Tabelle tab in die neue Tabelle tab‘;
  (Dabei ist es günstig, wenn Zeiger auf Datensätze eingesetzt werden. Dann müssen nämlich nur Zeiger und keine Datensätze kopiert werden.)
- Das alte Feld tab wird gelöscht und durch das neue ersetzt;

Problem:

Die insert-Operation, die zu einer Umorganisation der Hashtabelle führt, wird singulär sehr aufwendig. Wenn danach jedoch noch viele weitere insert-Operationen durchgeführt werden, wird der Aufwand jedoch amortisiert.

Problematisch kann der singulär erhöhte Aufwand bei einem interaktiven System sein.

Dynamische Hashverfahren (2)

Vergrößern der Tabelle mit verzögertem Umkopieren
(nur für Hashverfahren mit Verkettung)

Falls beim Einfügen ein bestimmter Füllungsgrad überschritten wird:

- Lege neue Tabelle tab‘ mit einer in etwa doppelten Größe m‘ an (m‘ sollte Primzahl sein).
- Bei jedem Zugriff auf die Tabelle (search, insert bzw. remove) wird zusätzlich der nicht-leere Tabelleneintrag (komplette verkettete Liste) mit kleinstem Index min in die neue Tabelle übertragen.
- Entscheide bei jedem Zugriff, ob Daten in alter oder neuer Tabelle stehen: Falls $h(k) = k \bmod m \leq min$, dann greife auf neue Tabelle tab‘ zu, sonst auf alte Tabelle tab.

  Beachte, dass bei alter Tabelle tab mit

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

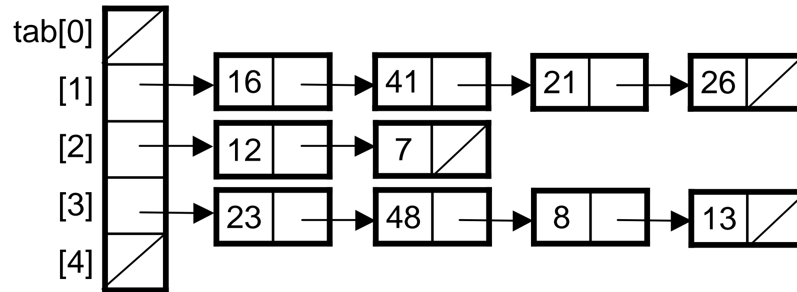
  und bei neuer Tabelle tab‘ mit

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

  zugegriffen wird.
- Falls alte Tabelle tab nicht mehr gebraucht wird (d.h. min = Größe der Tabelle tab), wird tab gelöscht und durch tab‘ ersetzt.

Dynamische Hashverfahren (3)

Beispiel:



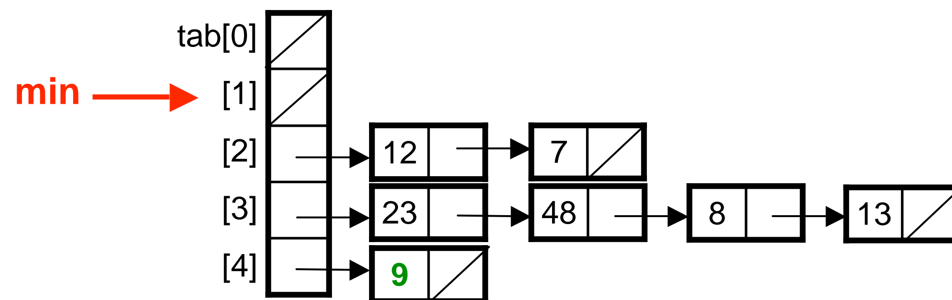
insert(9):

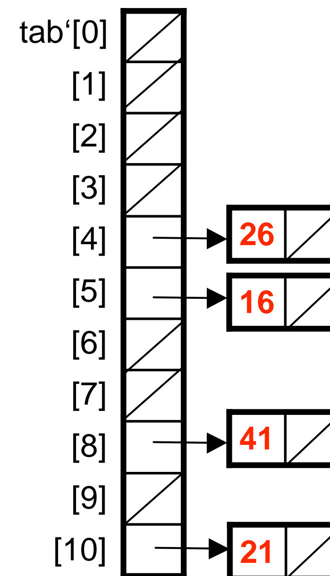
Lege neue Tabelle tab‘ der Größe 11 an, da Füllungsgrad von tab überschritten wird;

Übertrage kleinsten Tabelleneintrag tab[min] mit min = 1 nach tab‘;

Füge nun 9 in alte Tabelle ein

(da h(9) = 9 mod 5 = 4 > min).





Dynamische Hashverfahren (4)

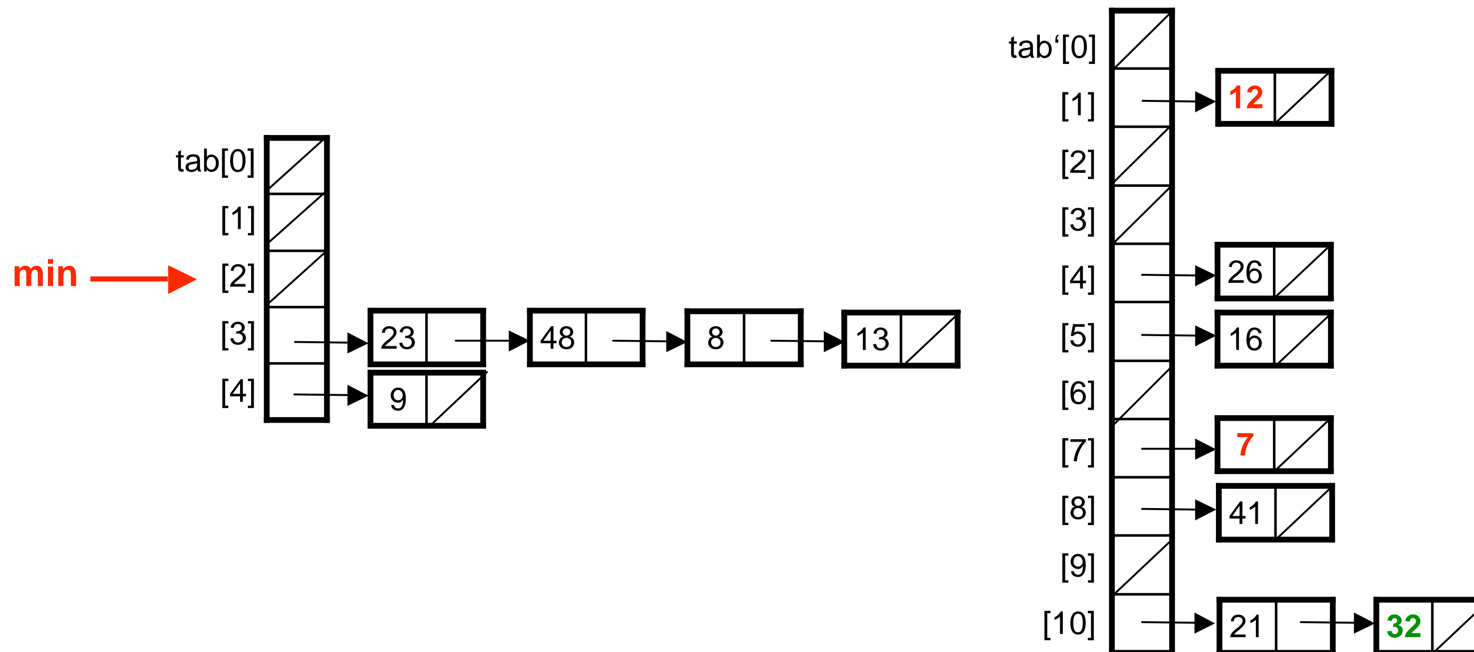
Beispiel (Fortsetzung):

insert(32):

Übertrage kleinsten Tabelleneintrag tab[min] mit min = 2 nach tab‘;

Füge nun 32 in neue Tabelle ein (h(32) = 32 mod 5 = 2 ≤ min).

Da h‘(32) = 32 mod 11 = 10 ist, wird 32 beim Index 10 eingetragen.



Anwendungen

- Compiler nutzen Hashtabellen zum Auffinden deklarierter Variablennamen (*symbol table*)
- Bei Graphen werden die Knoten oft über ihren Namen indexiert
- Spiele: Gemachte Spielzüge über momentane Position
- On-line Rechtschreibungskorrektur