Listen und Hashtabellen

Themenverzeichnis

Darstellung von Folgen

- Verkettete Listen
- Arrays
- Stapel und Warteschlangen

Hashtabellen

- Grundprinzip
- Hash-Funktionen / Qualität
- Handhabung von Kollisionen
- Zeitaufwand / Füllgrad
- Hashverfahren in Java

Darstellung von Folgen

Doppelt verkettete Listen

Jedes Glied (*Knoten*) einer doppelt verketteten Liste besitzt einen Zeiger auf seinen Vorgänger, einen Zeiger auf seinen Nachfolger und ein Objekt des generischen Typs *Element* mit dem eigentlichen Dateninhalt:

```
Class Handle = Pointer to Item

Class Item of Element                    // ein Knoten einer doppelt verketteten Liste
    e : Element                          //
    next : Handle
    prev : Handle
    invariant next→prev = prev→next = this
```

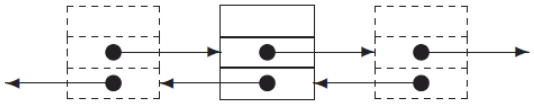


Abb. 3.1. Die Knoten einer doppelt verketteten Liste.

Es ist sehr vorteilhaft, wenn die Listen-Klasse einen „Dummyknoten“ (auch als *Anker* oder *Wurzel, Root, Header* bezeichnet) besitzt, der vorwärts auf das erste, rückwärts auf das letzte Kettenglied zeigt und selber keinen Dateninhalt hat:

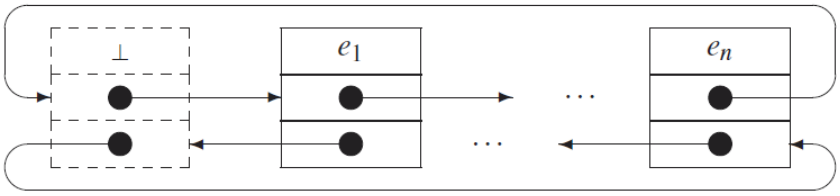


Abb. 3.2. Die Darstellung einer Folge $\langle e_1, \ldots, e_n \rangle$ durch eine doppelt verkettete Liste. Die Darstellung besteht aus $n+1$ ringförmig angeordneten Knoten, einem Dummyknoten h, der keinen Eintrag enthält, und einem Knoten für jeden der n Einträge der Folge. Der Knoten mit Eintrag e_i ist der Nachfolger des Knotens mit dem Eintrag e_{i-1} und der Vorgänger der Knotens mit dem Eintrag e_{i+1}. Der Dummyknoten sitzt zwischen dem Knoten mit dem Eintrag e_n und dem Knoten mit dem Eintrag e_1.

Der Dummyknoten ist auch in einer leeren Liste vorhanden. In diesem Fall zeigt er auf sich selbst als Vorgänger und Nachfolger.

Die meisten Manipulationen einer Liste können von der Operation *splice* abgeleitet werden. *splice* schneidet eine Teilliste heraus und fügt sie hinter einem bestimmten Zielknoten wieder ein:

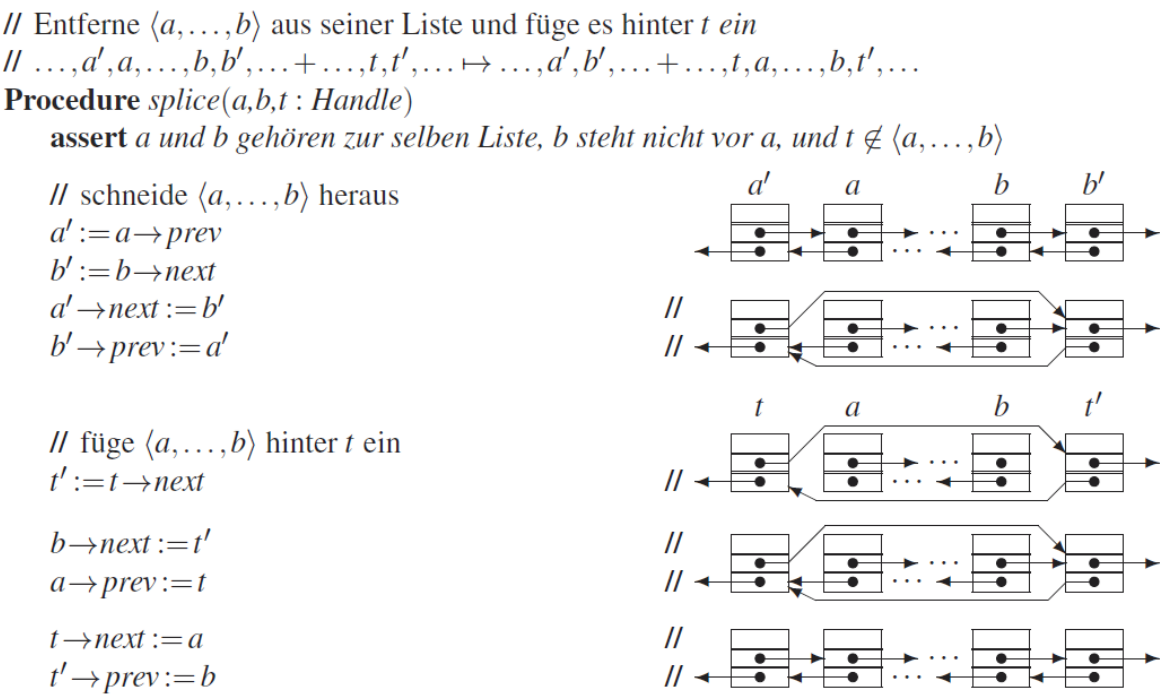


Abb. 3.3. Die Operation *splice* auf Listen.

Der Zielknoten *t* kann auch zu einer anderen Liste gehören.

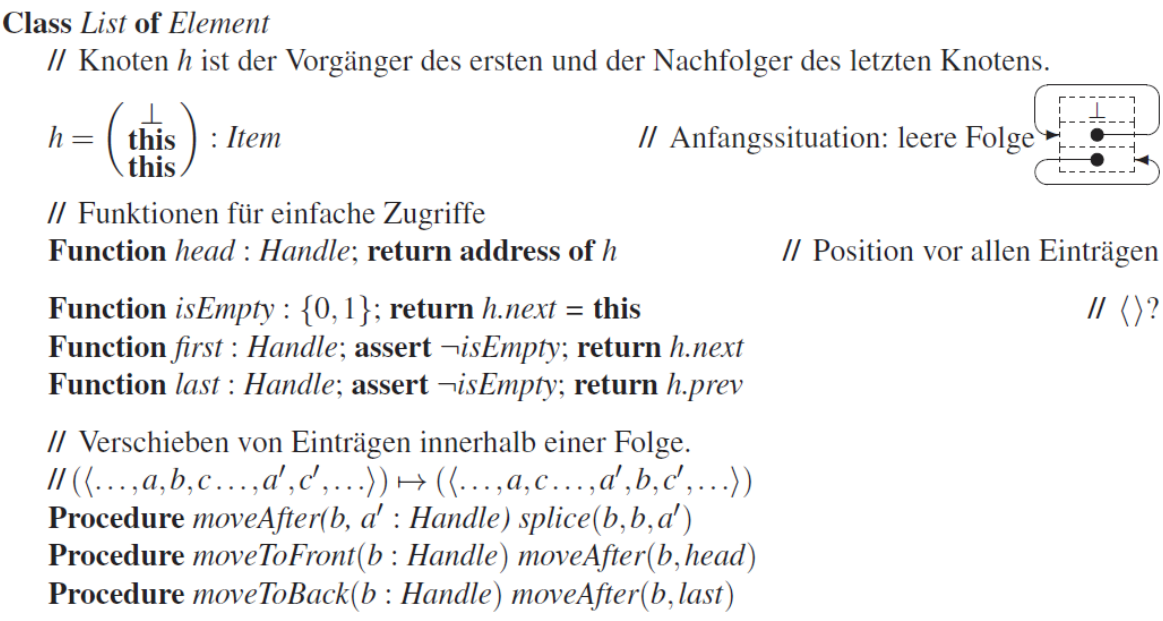


Abb. 3.4. Einige Operationen auf doppelt verketteten Listen (jeweils konstante Zeit).

Eine Optimierung von Listen-Manipulationen wird durch die Einführung der Liste *freeList* erreicht, die einen Vorrat von unbenutzten Knoten enthält und als Quelle von neu hinzufügenden Gliedern den anderen Listen dient. Sinngemäss werden die gelöschten Knoten in *freeList* entsorgt.
Bei Bedarf wird *freeList* mit dynamisch alloziertem Speicherplatz nachgefüllt, wodurch die Speicherplatz-Beschaffung für die einzelnen Knoten entfällt.

```
// Löschen und Einfügen von Einträgen.
// ⟨...,a,b,c,...⟩ ↦ ⟨...,a,c,...⟩
Procedure remove(b : Handle) moveAfter(b, freeList.head)
Procedure popFront remove(first)
Procedure popBack remove(last)

// ⟨...,a,b,...⟩ ↦ ⟨...,a,e,b,...⟩
Function insertAfter(x : Element; a : Handle) : Handle
    checkFreeList          // Stelle sicher, dass freeList nicht leer ist. Siehe auch Aufgabe 3.3
    a' := freeList.first                                  // Hole einen Knoten a',
    moveAfter(a', a)                                      // verschiebe ihn an die vorgesehene Stelle
    a' → e := x                                           // und trage den Eintrag x ein.
    return a'

Function insertBefore(x : Element; b : Handle) : Handle return insertAfter(e, pred(b))
Procedure pushFront(x : Element) insertAfter(x, head)
Procedure pushBack(x : Element) insertAfter(x, last)

// Bearbeitung ganzer Listen
// (⟨a,...,b⟩,⟨c,...,d⟩) ↦ (⟨a,...,b,c,...,d⟩,⟨⟩)
Procedure concat(L' : List)
    splice(L'.first, L'.last, last)

// ⟨a,...,b⟩ ↦ ⟨⟩
Procedure makeEmpty
    freeList.concat(this)
```

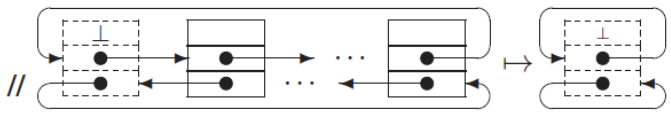


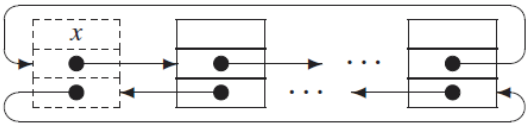
Abb. 3.5. Weitere Operationen auf doppelt verketteten Listen (jeweils konstante Zeit).

Die verketteten Listen zeichnen sich dadurch aus, dass die häufigsten Manipulationen (Einfügen, Löschen) mit einem konstanten Zeitaufwand durchführbar sind (vorausgesetzt, dass die Zielstelle der Mutation bereits bekannt ist).

Ein Suchverfahren kann hingegen nur als einfache Iteration (von einem Knoten zum nächsten) implementiert werden und hat demzufolge die Zeitkomplexität **O(n)** (eine binäre Suche ist auch für sortierte verkettete Listen nicht möglich).

Die Suche nach einem bestimmten Dateninhalt **x** kann erleichtert werden, indem der Dummyknoten vorübergehend mit diesem Dateninhalt versehen wird. Damit wird die Terminierung der Suche auch ohne zusätzliche Prüfungen sichergestellt (der Dummyknoten wird als „Wächterknoten“ benutzt):

Function *findNext*(x : *Element*; *from* : *Handle*) : *Handle*
    $h.e = x$ // Wächterknoten
    while $from \rightarrow e \neq x$ **do**
        $from := from \rightarrow next$
    $h.e := \perp$
return *from*



Eine andere Abweichung vom konstanten Zeitaufwand kann entstehen, wenn die Länge der Liste als Attribut mitgeführt wird und Teilfolgen von mehreren Knoten hinzugefügt oder gelöscht werden.

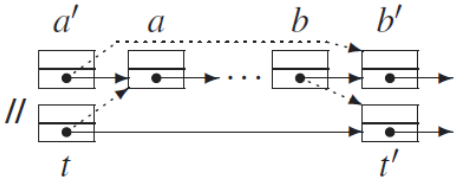
Einfach verkettete Listen

In dieser „schlankeren“ Variante besitzt ein Knoten nur den Vorwärts-Zeiger, wodurch die Pflege der Liste bei Manipulationen reduziert, die Benutzung jedoch erschwert wird.
Der Operation *splice* muss z.B. der Vorgänger der manipulierten Teilliste als Parameter mitgegeben werden, damit die entstandene Lücke wieder geschlossen werden kann:

$$// \,(\langle \ldots, a', a, \ldots, b, b' \ldots \rangle, \langle \ldots, t, t', \ldots \rangle) \mapsto (\langle \ldots, a', b' \ldots \rangle, \langle \ldots, t, a, \ldots, b, t', \ldots \rangle)$$

Procedure *splice*(a', b, t : *SHandle*)

$$\begin{pmatrix} a' \rightarrow next \\ t \rightarrow next \\ b \rightarrow next \end{pmatrix} := \begin{pmatrix} b \rightarrow next \\ a' \rightarrow next \\ t \rightarrow next \end{pmatrix}$$



Arrays

Im Gegensatz zu verketteten Listen sind in Arrays die Elemente nummeriert und es wird auf ein bestimmtes Element via seine Nummer (Index) zugegriffen, wobei diesbezüglicher Zeitaufwand nicht davon abhängt, wie weit das Element von der vorher besuchten Stelle der Folge entfernt ist. In diesem Sinne ist der Zugriff auf Array-Elemente wahlfrei (==> *random access*).

Statische Arrays : Die Länge ist im Voraus bekannt und unveränderlich.

Dynamische (unbeschränkte) Arrays: Die Länge ist variabel, kann jederzeit beliebig geändert werden.

Im Folgenden werden einige Aspekte von unbeschränkten Arrays thematisiert. Die Implementierung von den folgenden Operationen wird analysiert:

$$\begin{aligned} \langle e_1, \ldots, e_n \rangle .pushBack(e) &= \langle e_1, \ldots, e_n, e \rangle \ , \\ \langle e_1, \ldots, e_n \rangle .popBack &= \langle e_1, \ldots, e_{n-1} \rangle \qquad (\text{für } n \geq 1) \ , \\ size(\langle e_1, \ldots, e_n \rangle) &= n \ . \end{aligned}$$

Ein mögliches Schema der Realisierung sieht vor, dass die **n** Einträge in einem Puffer der festen Grösse **w** gehalten werden, wobei stets gilt **w >= n** .
Wenn der Puffer durch Hinzufügungen voll wird (**n = w**), löst die nächste *pushBack*-Hinzufügung einen *reallocate* – Aufruf aus, bei dem ein neuer Puffer der Grösse **2w** alloziert, die bestehenden Einträge in diesen neuen Puffer kopiert und der alte Puffer freigegeben wird. Anschliessend kann die auslösende *pushBack*-Hinzufügung zum neuen Puffer erfolgen.
Um einen angemessenen Speicherplatz-Gebrauch zu bewahren wird *reallocate* auch dann aufgerufen, wenn die Anzahl von gespeicherten Einträgen infolge von *popBack*-Löschungen auf einen Viertel der Puffergrösse sinkt (**n <= w/4**). In diesem letzteren Fall wird die Puffergrösse auf **w/2** reduziert.

Das oben beschriebene Schema kann in Pseudocode wie folgt dargestellt werden:

```
Class UArray of Element
    Constant β = 2 : ℝ>0                                   // Wachstumsfaktor
    Constant α = 4 : ℝ>0                                   // Speicherüberhang im schlechtesten Fall
    w = 1 : ℕ                                              // bereitgestellte Arraygröße
    n = 0 : ℕ                                              // benutzte Arraygröße
    invariant n ≤ w < αn or n = 0 and w ≤ β
    b : Array [1..w] of Element                            //  1      n      w
                                                           //  b → | e1 | ··· | en |  | ··· |  |

    Operator [i : ℕ] : Element
        assert 1 ≤ i ≤ n
        return b[i]

    Function size : ℕ     return n

    Procedure pushBack(e : Element)                        // Beispiel für n = w = 4:
        if n = w then                                      //  b → | 1 | 2 | 3 | 4 |
            reallocate(βn)                                 //  b → | 1 | 2 | 3 | 4 |  |  |  |  |
        b[n+1] := e                                        //  b → | 1 | 2 | 3 | 4 | e |  |  |  |
        n++                                                //  b → | 1 | 2 | 3 | 4 | e |  |  |  |

    Procedure popBack                                      // Beispiel für n = 5, w = 16:
        assert n > 0                                       //  b → | 1 | 2 | 3 | 4 | 5 |  ...  |
        n--                                                //  b → | 1 | 2 | 3 | 4 | 5 |  ...  |
        if αn ≤ w ∧ n > 0 then                             //  verringere überflüssigen Speicherplatz
            reallocate(βn)                                 //  b → | 1 | 2 | 3 | 4 |  |  |  |  |

    Procedure reallocate(w' : ℕ)                           // Beispiel für w = 4, w' = 8:
        w := w'                                            //  b → | 1 | 2 | 3 | 4 |
        b' := allocate Array [1..w'] of Element            //  b' → |  |  |  |  |  |  |  |  |
        (b'[1],...,b'[n]) := (b[1],...,b[n])               //  b' → | 1 | 2 | 3 | 4 |  |  |  |  |
        dispose b                                          //  b → | 1 | 2 | 3 | 4 |  (crossed out)
        b := b'                                            //  Umstellen des Zeigers b → | 1 | 2 | 3 | 4 |  |  |  |  |
```

Abb. 3.6. Pseudocode für unbeschränkte Arrays.

Man kann zeigen, dass der Aufbau eines Arrays mit einer beliebigen Abfolge von **m** *pushBack/popBack* – Aufrufen (ausgehend von einem leeren Array und das obige Schema befolgend) einen Zeitaufwand der Komplexität **O(m)** beansprucht. Somit können die einzelnen Aufrufe als Operationen mit konstantem Aufwand betrachtet werden.

Zu diesem Zweck wird angenommen, dass die Verschiebung eines Eintrages von einem Speicherort zum anderen eine konstante Zeitspanne (1 „Jeton“) beansprucht. Die Zeitaufwände in der untenstehenden Tabelle sind in Jetons angegeben.
Der Gesamtaufwand der Abfolge wird mit einem fiktiven Aufwand **3m** verglichen, der entstehen würde, wenn jede Operation genau **3** Jetons beanspruchen würde. Dieser **3m** – Aufwand wird hier als *Majorante* (obere Grenze) benutzt: wenn der tatsächliche Aufwand durchgehend kleiner oder gleich bleibt, als eine Majorante der Komlexität **O(m)**, ist damit die **O(m)** – Komplexität des Array-Aufbaus bewiesen.

m	Operation	realloc.	n_vor	n_nach	w_vor	w_nach	Aufwand	Aufw_ges	3m	3m – Aufw_ges
1	pushBack	N	0	1	1	1	1	1	3	2
2	pushBack	J	1	2	1	2	2	3	6	3
3	pushBack	J	2	3	2	4	3	6	9	3
4	pushBack	N	3	4	4	4	1	7	12	5
5	pushBack	J	4	5	4	8	5	12	15	3
6	pushBack	N	5	6	8	8	1	13	18	5
7	pushBack	N	6	7	8	8	1	14	21	7
8	pushBack	N	7	8	8	8	1	15	24	9
9	pushBack	J	8	9	8	16	9	24	27	3
10	pushBack	N	9	10	16	16	1	25	30	5
11	pushBack	N	10	11	16	16	1	26	33	7
12	pushBack	N	11	12	16	16	1	27	36	9
13	pushBack	N	12	13	16	16	1	28	39	11
14	pushBack	N	13	14	16	16	1	29	42	13
15	pushBack	N	14	15	16	16	1	30	45	15
16	pushBack	N	15	16	16	16	1	31	48	17
17	pushBack	J	16	17	16	32	17	48	51	3
18	popBack	N	17	16	32	32	0	48	54	6
19	popBack	N	16	15	32	32	0	48	57	9
20	popBack	N	15	14	32	32	0	48	60	12
21	popBack	N	14	13	32	32	0	48	63	15
22	popBack	N	13	12	32	32	0	48	66	18
23	popBack	N	12	11	32	32	0	48	69	21
24	popBack	N	11	10	32	32	0	48	72	24
25	popBack	N	10	9	32	32	0	48	75	27
26	popBack	J	9	8	32	16	8	56	78	22

Aus der letzten Spalte der obigen Tabelle ist ersichtlich, dass die aufwändigen *reallocate* – Aufrufe nicht zu negativen Differenzen **(3m – Aufw_ges)** führen, weil sich dieser Wert in den vorherigen Schritten genügend aufgebaut hat, um den negativen Ausschlag auszugleichen.
Mit anderen Worten: der *reallocate* – Aufwand wird mit den „Ersparnissen“ der Vorherigen Schritte im Voraus „amortisiert“.
In diesem Sinne spricht man von *amortisiert konstanter Rechenzeit* einer Operation.
Solche Aussage ist sehr ähnlich wie „die Rechenzeit ist im Durchschnitt konstant“, hat aber nicht die genau gleiche Bedeutung, weil sich die Amortisierung nur auf eine spezifische Abfolge bezieht, nicht auf eine beliebige, zufällig selektierte Probe (wie der Durchschnitt).

Das Ergebnis der obigen Darstellung lässt sich unschwer erweitern auf den allgemeinen Fall von beliebig langen Abfolgen, bei welchen nur eine Art von Operation (*pushBack*- oder *popBack)* zwischen zwei *reallocate*-Aufrufen vorkommt. Im Fall von alternierenden Abfolgen sind die Kosten einer Operation niedriger, weil der *reallocate*-Aufwand auf mehr Einzeloperationen verteilt wird. Die oben untersuchten zusammenhängenden Abfolgen einer Art stellen somit den schlechtesten Fall dar.

Stapel und Warteschlangen

Es handelt sich um die vorher beschriebenen Datenstrukturen (Listen oder Arrays), welche in einer eingeschränkten Art benutzt werden.

- Stapel (Stack) : LIFO – Schema (Last In First Out)
- Warteschlange (Queue) : FIFO-Schema (First In First Out)
- Doppelschlange (Deque) : Zuwachs und Abbau auf beiden Seiten

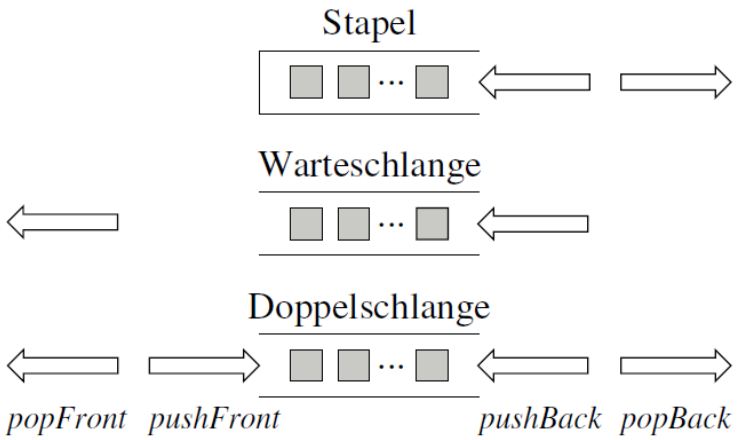
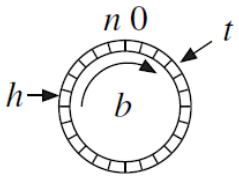


Abb. 3.7. Operationen auf Stapeln, Warteschlangen und Doppelschlangen (Deques).

Beispiel: Warteschlange, implementiert mit einem statischen Array (Zirkulär-Puffer)




```
Class BoundedFIFO(n : ℕ) of Element
    b : Array [0..n] of Element
    h = 0 : ℕ                          // Index des ersten Eintrags
    t = 0 : ℕ                          // Index der ersten freien Position

    Function isEmpty : {0,1}; return h = t

    Function first : Element; assert ¬isEmpty; return b[h]

    Function size : ℕ; return (t − h + n + 1) mod (n + 1)

    Procedure pushBack(x : Element)
        assert size < n
        b[t] := x
        t := (t + 1) mod (n + 1)

    Procedure popFront assert ¬isEmpty; h := (h + 1) mod (n + 1)
```

Abb. 3.8. Eine Implementierung von beschränkten Warteschlangen mit Arrays.

Performance-Vergleich

Tabelle 3.1. Rechenzeiten für Operationen auf Folgen mit n Einträgen. Man muss sich jeden Tabelleneintrag mit einem „ $O(\cdot)$“ eingefasst denken. *List* steht für doppelt verkettete Listen, *SList* steht für einfach verkettete Listen, *UArray* für unbeschränkte Arrays und *CArray* für zyklisch organisierte Arrays.

Operation	*List*	*SList*	*UArray*	*CArray*	Erläuterungen („*“)
$[\cdot]$	n	n	1	1	
size	1*	1*	1	1	nicht mit *splice* für mehrere Listen
first	1	1	1	1	
last	1	1	1	1	
insert	1	1*	n	n	nur für *insertAfter*
remove	1	1*	n	n	nur für *removeAfter*
pushBack	1	1	1*	1*	amortisiert
pushFront	1	1	n	1*	amortisiert
popBack	1	n	1*	1*	amortisiert
popFront	1	1	n	1*	amortisiert
concat	1	1	n	n	
splice	1	1	n	n	
findNext, ...	n	n	n^*	n^*	cache-effizient

Hashtabellen

Hashtabellen sind eine Form der Datenorganisation, welche eine effiziente Suche und Verwaltung ermöglicht.
Die damit verwalteten Dateninhalte sind *assoziative Arrays* ==> Sammlungen von Paaren { *Schlüssel / Datensatz*}.
Als Beispiel eines assoziativen Arrays kann ein Telefonbuch dienen: der Name des Abonnenten ist der Schlüssel, die Telefonnummer ist der damit verknüpfte Datensatz.

Operationen, welche für die Verwaltung eines assoziativen Arrays **S** notwendig sind:

- $S.build(\{e_1,\ldots,e_n\})$: $S := \{e_1,\ldots,e_n\}$.
- $S.insert(e : Element)$: Falls es ein $e' \in S$ mit $key(e') = key(e)$ gibt: $S := (S \setminus \{e'\}) \cup \{e\}$; andernfalls: $S := S \cup \{e\}$.
- $S.remove(x : Key)$: Falls es ein $e \in S$ mit $key(e) = x$ gibt: $S := S \setminus \{e\}$.
- $S.find(x : Key)$: Falls es ein $e \in S$ mit $key(e) = x$ gibt, dann gib e zurück, andernfalls gib $\perp$ zurück.

Typisches Problem: Personaldaten von Angestellten einer Firma speichern mit Schlüssel = AHV-Nummer so, dass beim Zugriff auf eine bestimmte Person nicht durch alle iteriert werden muss ➔ wahlfreier Zugriff (*random access*).

Beispiel AHV-Nummer: 756.7496.1209.00
(➔ ohne Punkte eine 13-stellige Zahl)

Theoretisch könnte man ein Array benutzen, in dem die AHV-Nummer als Index dient. Damit wäre der wahlfreie Zugriff gewährleistet um den Preis einer massiven Speicherplatz-Verschwendung.

Lösungsansatz: Speicherung von beliebigen Zahlen in einem statischen Array, wobei der Index jeder mit Zahl mit einer Formel (Hashfunktion) zugewiesen wird.

- Feld von 0 bis 9; Hashfunktion $h(i) = i \bmod 10$
- Feld nach Einfügen von 42 und 119:

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Möglichkeit der *Kollision*: Versuch des Eintragens von 69

Die Güte einer Hashfunktion ist eine möglichst gleichmässige Streuung der berechneten Index-Werte.

Grundlagen: Hashfunktionen

- hängen vom Datentyp der Elemente und konkreter Anwendung ab
- für Integer oft

$$h(i) = i \bmod N$$

  (funktioniert gut wenn N eine Primzahl ist)
- für andere Datentypen: Rückführung auf Integer
  - Fließpunkt-Zahlen: Addiere Mantisse und Exponent
  - Strings: Addiere ASCII/Unicode-Werte der/einiger Buchstaben, evtl. jeweils mit Faktor gewichtet

$$h(s) = (a_0 \cdot s[0] + a_1 \cdot s[1] + \cdots + a_{l-1} \cdot s[l-1]) \bmod N$$

Universelles Hashing ist eine theoretisch optimierte Methode der Ermittlung von möglichst kollisionsfreien Hash-Werten.

Beispiel: Die vorher erwähnte AHV-Nummer 756.7496.1209.00 soll in einer Hashtabelle der Länge 100 gespeichert werden. Dazu wird diese Nummer zerteilt in Gruppen von Ziffern, die als Zahlen kleiner als 100 sind, d.h. aus maximal zwei Ziffern bestehen:

$$756.7496.1209.00$$

Aus den so entstandenen Teilzahlen wird eine gewichtete Summe gebildet, wobei die Gewichtungsfaktoren $\mathbf{a_1}$ .. $\mathbf{a_7}$ zufällig gewählt und ebenfalls kleiner als 100 sind:

$$S = a_1 * 7 + a_2 * 56 + a_3 * 74 + a_4 * 96 + a_5 * 12 + a_6 * 9 + a_7 * 0$$

Den Hash-Wert **h** ermittelt man dann via Modulo-Division mit der grössten Primzahl unter 100:

$$h = S \bmod 97$$

(damit bleiben die letzten 4 Tabellenplätze unbenutzt).

Die Gewichtungsfaktoren $\mathbf{a_1}$ .. $\mathbf{a_7}$ sind prinzipiell frei wählbar, in praktischen Situationen ist jedoch sinnvoll, die wenig variierten Gruppen wenig zu gewichten (hier sind es die ersten 2 Gruppen, welche CH-Landcode darstellen).

Mit $a_1 = 1$, $a_2 = 1$, $a_3 = 59$, $a_4 = 13$, $a_5 = 91$, $a_6 = 68$, $a_7 = 29$ erhalten wir

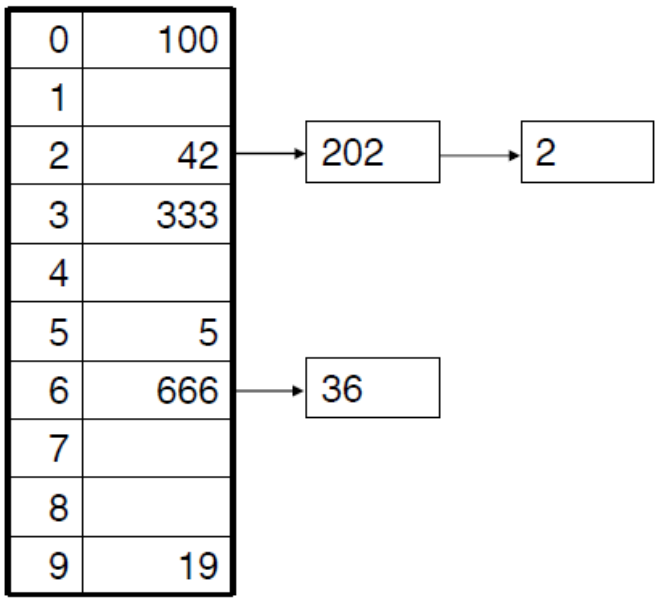
$$h = 7381 \bmod 97 = 9$$

Die Kollisionen werden mit zwei verschiedenen Ansätzen gehandhabt:

- Verkettung
- Sondierung

Verkettung der Überläufer

- Idee: alle Einträge einer Array-Position werden in einer verketteten Liste verwalten (hier: N=10)



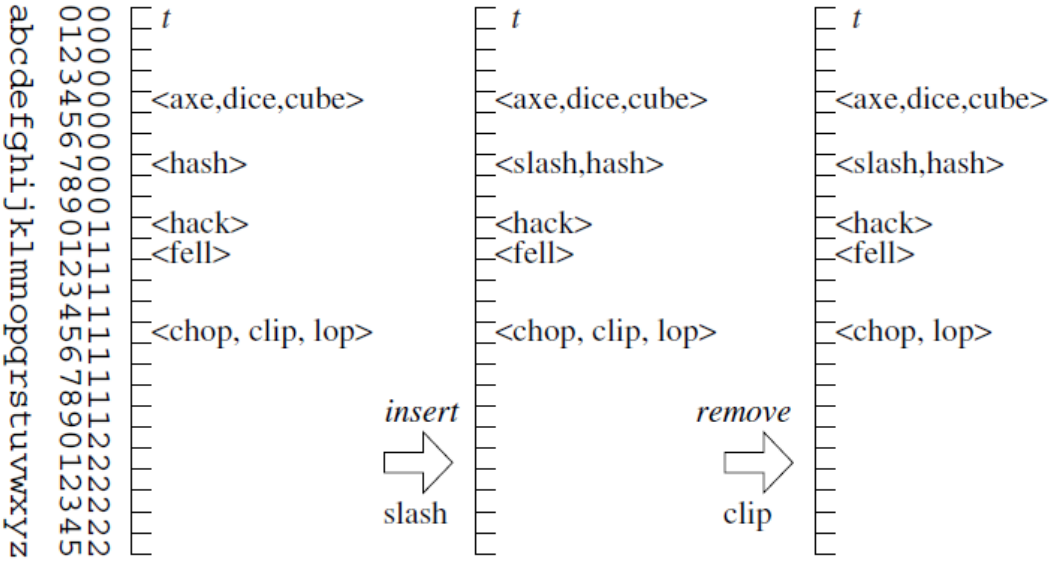


Abb. 4.1. Hashing mit Verkettung. Wir verwenden ein Array t von Folgen. In dem in der Abbildung dargestellten Beispiel ist in der Datenstruktur eine Menge von Wörtern (kurze Synonyme von „hash“) gespeichert, mittels einer Hashfunktion, die ein Wort auf die Position seines letzten Buchstabens im Alphabet abbildet, d. h. auf einen Wert in 0..25. Offensichtlich ist dies keine besonders gute Hashfunktion.

Lineares Sondieren /1

- Falls Eintrag $T[h(e)]$ in Hashtabelle T besetzt ist, teste nach freien Positionen in linearer Reihenfolge:

  $T[h(e) + 1]$, $T[h(e) + 2]$, ..., $T[h(e) + i]$, ...

  (Genau genommen: $T[(h(e) + i) \bmod N]$

- *Anmerkung:* Sondieren muss bei Einfügen und Suchen erfolgen!

Lineares Sondieren /2

h(*i*)	leer	Insert 89	Insert 18	Insert 49	Insert 58	Insert 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Hashfunktion: $h(i) = i \bmod 10$

Das Löschen von Einträgen ist beim Sondierung-Ansatz nicht unproblematisch, weil die „Sondierungsketten“ dadurch unterbrochen werden und das Ende der Kette nicht mehr erreichbar ist.
Eine mögliche Abhilfe besteht darin, die Einträge als ungültig zu markieren ohne sie zu entfernen. Bei einer Anhäufung von ungültigen Einträgen ist es sinnvoll, die Tabelle neu aufzubauen.
Es ist natürlich auch möglich, die unterbrochenen Ketten sofort nach dem Entstehen zu reparieren (eleganter aber aufwändiger):

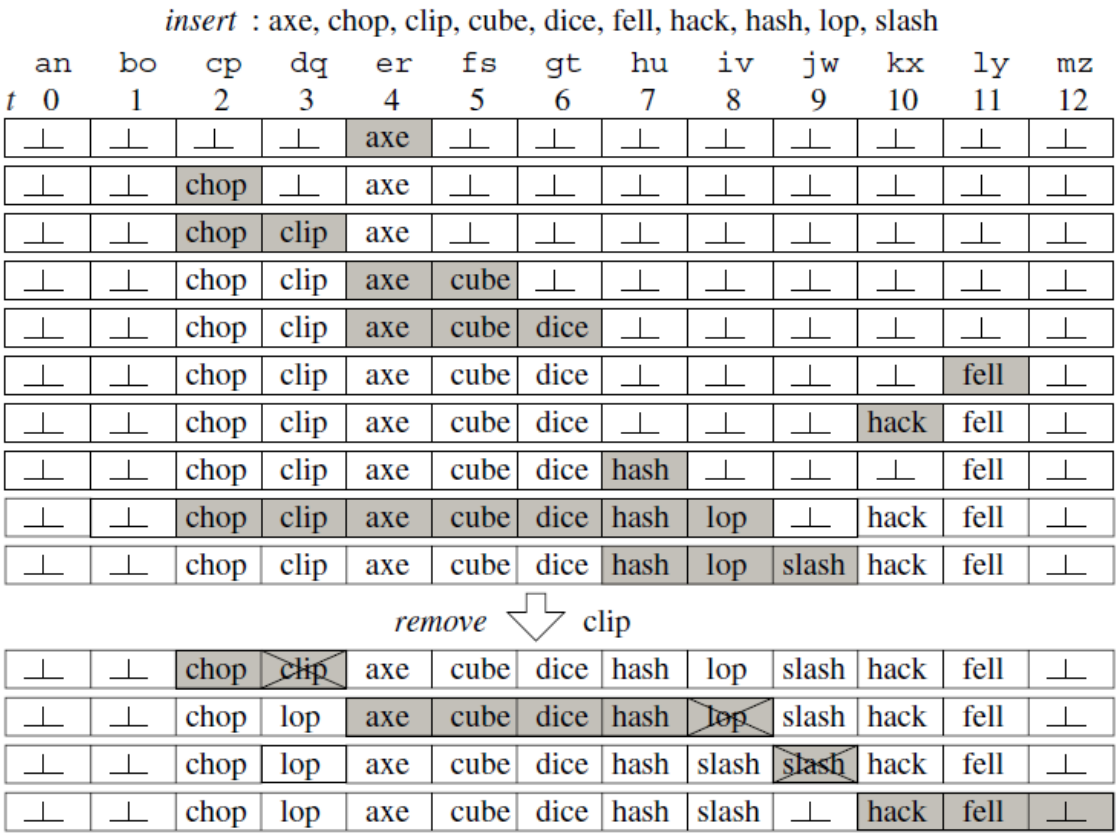


Abb. 4.2. Hashing mit linearem Sondieren. In einer Tabelle *t* mit 13 Plätzen werden Synonyme von “(*to*) *hash*” gespeichert. Die Hashfunktion bildet den letzten Buchstaben des Wortes auf die Zahlen 0..12 ab wie oben im Bild angedeutet: „a“ und „n“ werden auf 0 abgebildet, „b“ und „m“ auf 1 usw. Die Wörter werden in alphabetischer Reihenfolge nacheinander eingefügt. Dann wird das Wort „clip“ entfernt. Im Bild ist angegeben, wie sich mit jeder Operation der Zustand der Tabelle ändert. Bereiche, die bei der Ausführung der aktuellen Operation durchmustert werden, sind grau unterlegt.

Quadratisches Sondieren

- Problem: Lineares Sondieren neigt zur „Klumpenbildung“
- Idee: falls $T[h(e)]$ besetzt ist, versuche

  $T[h(e) + 1]$, $T[h(e) + 4]$, $T[h(e) + 9]$, …, $T[h(e) + i^2]$, …
- Variante:

  $T[h(e) + 1]$, $T[h(e) - 1]$, $T[h(e) + 4]$, $T[h(e) - 4]$, …
- ! Anfällig gegenüber falsch gewähltem N
  - Beispiel $N = 16$:

    $4^2 \bmod 16 = 0 = 0^2$, $5^2 \bmod 16 = 9 = 3^2$,

    $6^2 \bmod 16 = 4 = 2^2$, $7^2 \bmod 16 = 1 = 1^2$,…

    → wähle Primzahlen für N

Quadratisches Sondieren /2

	leer	insert 89	Insert 18	Insert 49	Insert 58	Insert 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Der Zeitaufwand ist beim Hashverfahren stark vom Füllgrad abhängig:

Füllgrad = [Anzahl gespeicherter Einträge] / [Anzahl Speicherplätze (Buckets)]

Eine allgemein anwendbare Grobschätzung besagt, dass sich die Rechenzeit von Lese- und Schreibzugriffen wie **(1 + Füllgrad)** verhält:

Rechenzeit ~ (1 + Füllgrad)

Diese Schätzung ist gut nachvollziehbar für geringe (<< 1) wie auch hohe Füllgrade (> 1). Für sehr kleine Füllgrade tendiert die Komplexität einer Operation zu **O(1)** (konstanter Zeitaufwand, unabhängig von der gespeicherten Menge), bei überfüllten Hashtabellen degeneriert die Komplexität bezüglich Anzahl gespeicherter Einträge zu **O(n)**.

Aufwand beim Hashen

- bei geringer Kollisionswahrscheinlichkeit:
  - Suchen in $O(1)$
  - Einfügen in $O(1)$
  - Löschen bei Sondierverfahren:
    - nur Markieren der Einträge als gelöscht: $O(1)$
    - „Rehashen“ der gesamten Tabelle notwendig: $O(n)$
- Füllgrad über 80 % : Einfüge- / Suchverhalten wird schnell dramatisch schlechter aufgrund von Kollisionen (beim Sondieren)

Aufwand beim Hashen (Verkettete Überläufer)

- Absoluter Füllgrad $\beta = k / N$
  - k Anzahl gespeicherter Elemente
  - N Anzahl Buckets
- erfolglose Suche (Hashing mit Überlaufliste): $1+\beta$ Suchschritte
- erfolgreiche Suche ebenfalls in dieser Größenordnung

Aufwand beim Hashen (Sondieren detailliert)

- Relativer Füllgrad (load factor) $\alpha = m / N$
  - m Anzahl belegter Buckets, N Anzahl Buckets
- Wahrscheinlichkeiten basieren auf Füllgrad α
  - Bucket belegt: Wahrscheinlichkeit α
  - Bucket belegt und sondiertes Bucket auch belegt: ungefähr α^2 etc.
- Somit kann Anzahl Suchschritte abgeschätzt werden durch: $1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \frac{1}{1-\alpha}$
- erfolgreiche Suche besser (keine Suche bis zu freiem Platz), aber gleiche Größenordnung

Aufwand beim Hashen (Sondieren)

Beispiele:

- Aufwand beim Suchen: $\frac{1}{1-\alpha}$

- typische Werte:

$$\frac{1}{1-0.5} = 2 \qquad \frac{1}{1-0.9} = 10$$

Hashverfahren in Java

Der Einsatz von Hashverfahren in Java kann drei verschiedene Konzepte befolgen.

1. *Konzept: Standard-Klassen benutzen*

Datenstrukturen in `java.util`

`Hashtable<K,V>`: Array-basierte Implementierung einer Hash-Tabelle mit verketteter Liste als Kollisionsbehandlung, Re-Hashing und Steuerung der Tabellengröße und maximalen `loadFactor` (effizient bis 0.75).

`HashMap<K,V>`: ähnlich Hashtable, erlaubt aber `null` als Schlüssel.

`LinkedHashMap<K,V>`: verwaltet zusätzlich eine doppelt verkettete Liste zum Erhalten der Einfügereihenfolge.

`HashSet<E>`: `HashMap`-basierte Implementierung einer Menge mit $O(1)$ Aufwand für Lookup.

`LinkedHashSet<E>`: LinkedHashMap-basierte Implementierung einer Menge.

`IdentityHashMap<K,V>`: Hash-Tabelle unter Verwendung der Objektreferenz statt des HashCodes.

`WeakHashMap<K,V>`: Hash-Tabelle, aus der gelöschte Elemente als Referenz automatisch gelöscht werden.

Die obigen Klassen sind generisch, parametriert mit *<K, V>* (Key, Value). Mit *String*, *Integer* usw. als *K* können die Standard-Klassen ohne jeglichen Zusatz in einer Anwendung benutzt werden.

2. *Konzept: Methoden hashCode() und equals() der Klasse Object überschreiben*

- Vordefinierte Methode zur Berechnung eines Hashwertes:

  `int java.lang.Object.hashCode()`
- kann überschrieben werden, sollte folgende Eigenschaften haben:
  - Bei jedem Aufruf für ein Objekt gleicher Wert
  - Gleiche Objekte (`x.equals(y)=true`) liefern gleichen Hashwert
- Default-Implementierung in Java:
  - Von Speicheradresse des Objektes abgeleitet
  - Für `java.lang.String`:

    $h(s) = s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$

Dieses Vorgehen kommt zum Einsatz, wenn eine Anwenderklasse als Parameter-Klasse *K* (Key) eingesetzt wird. In diesem Fall muss via Überschreibung die Gleichheit von zwei *K*-Objekten im Sinne der Anwendung definiert werden (sonst werden *Object* – Methoden benutzt ➔ referentielle Gleichheit).

3. *Konzept: Anwender-Klassen mit Hash-Zugriff*

Bei der Entwicklung von solchen Klassen ist es von Vorteil, wenn sie die *interface Hashing* implementieren (➔ die Methoden *add()* und *contains()* muss dann die Klasse zur Verfügung stellen).

Schnittstelle für Hashtabellen

```
interface Hashing {
   void add(Object o)
      throws HashTableOverflowException;
   boolean contains(Object o);
}
```

- Einfügen eines Objektes: `add`
- Test auf Enthaltensein (Auslesen): `contains`
- Hashfunktion: `Object.hashCode()`

Beispiel:

Implementierung einer Hashtabelle /1

```
public class LinkedHashTable
      implements Hashing {
   LinkedList[] table; // Feld mit Listen

   public LinkedHashTable(int size) {
      // Feld aufbauen
      table = new LinkedList[size];
   }
   …
}
```