Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Tabellen

Überblick in die Einführung einfachster Abstrakter Datentypen mit Tabellen:

1. Lineare Arrays als Datenstrukturen
2. Der Stapelspeicher (LIFO)
3. Die Warteschlange (FIFO)
4. Die Hashtabelle

Lineare Arrays

Ein Array ist eine Tabelle mit einer Spalte und *n* Zeilen.

- Ein Array ist eine monosortige lineare Folge von Werten und somit Speicherzellen.
  - Jede Speicherzelle hat den gleichen Datentpy, z.B. skalare Werte wie Ganzzahlen, aber acuh Datenstrukturen
  - Die Größe des Arrays ist fest (*n*)
  - Jede Zelle eines Arrays wird über einen eindeutigen Index adressiert, i.A. immer mit 0 beginnend (erstes Element) bis *n*-1 (letztes Element):

$$T = \begin{pmatrix} v_0 \\ v_1 \\ .. \\ v_{n-1} \end{pmatrix}$$

$$Addr = I(T) = \{0, 1, 2, .., n-1\}$$

Operationen

1. Elementare (eingbaute) Operationen:
   - Lesen eines Elements `T[index]`
   - Schreiben eines Elements `T[index]=ε`
   - **RAM Modell**: freier Indexzugriff in beliebiger Reihenfolge!
2. Abstrakte Operationen:
   - Suche: Suche den Index eines Elementwertes
   - Sortieren: Sortieren der Werte in einer definierten Reihenfolge
   - usw.

Es gibt beliebig viele Operationen die man auf Tabllen anwenden und implementieren kann. Suche und Sortierung (kommt noch) sind die wichtigsten.

3. Linear Arrays = Mathematische Vektoren
   - Numerik
   - Vektorarithmetik

Schlüssel-Wert Paare

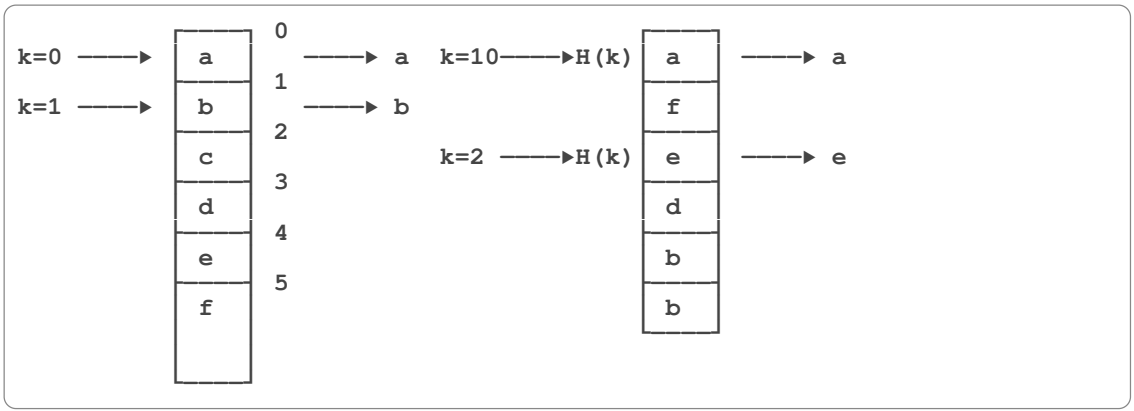
Der Index einer Tabelle ist ein Schlüssel zur Erlangung eines Werts. So könnte man numerischen Indexwerten eine Zecihenkette zuordnen.

```
T=['Zero','One','Two']
print(T[0])
```

Bsp. 1. Schlüssel-Wert Paare für Lookup-Tabellen

- Der große Vorteil: "Suchzeit", d.h. Abbildung eines Schlüssels (Index) auf einen Wert geschieht in konstanter Laufzeit O(1)!
- Der große Nachteil: Die Schlüsselfolge ist fest vorgegeben und startet bei 0. Bei dünn besetzten Schlüsselmengen bräuchte man dünnbesetzte riesig Tabellen!
    - Daher werden noch kompaktere Hashtabellen eingeführt.

- Lineare Tabellen verwenden eine lineare Suchfunktion für Schlüssel, i.A. $i=H(k)=k$
- Hashtabellen verwenden eine "andere" Suchfunktion $H(k)$ die beliebige Schlüsselverteilungen (theoretisch) auf einen Index i abbilden.



Def. 1. Lineare versa Hashtabellen als Suchtabellen (Look-up tables)

Der Stapelspeicher

- Ein Kellerspeicher oder auch Stapelspeicher alias Stack genannt ist eine lineare Tabelle (Array), aber
  - Es gibt aber keine freie Indizierung (kein RAM Modell)
  - Er implementiert aber Operationen mit einer Last-In First-Out Reigenfolge
- Nur abstrakte Operationen:
  - **Push**: Lege einen neuen Wert auf den Stapel ab (Last-In)
  - **Pop**: Holt den obersten Wert vom Stapel (Last-Out)

Der Stapelspeicher

```
type Stack(T)
operators
  empty : → Stack
  new : n → Stack(n)
  push : Stack × T → Stack
  pop : Stack → T
  top : Stack → T
  is_empty : Stack → Bool
  is_full : Stack → Bool
axioms ∀s : Stack, ∀x : T
  pop (push (s, x)) = x
  top (push (s, x)) = x
  is_empty (empty) = true
  is_empty (push (s, x)) = false
  is_full (empty) = false
  is_full (n*push (s, x)) = true
```

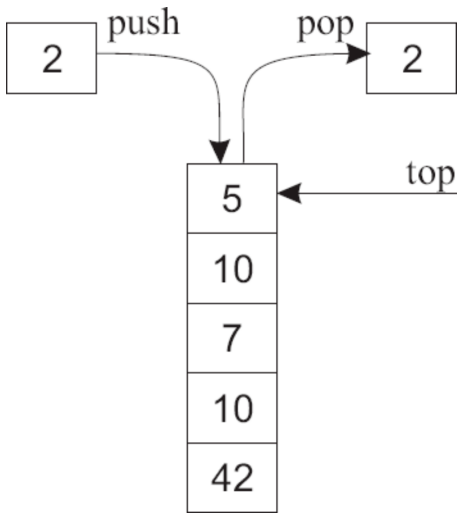
Def. 2. ADT-Spezifikation eines Stacks

Der Stapelspeicher

LIFO Prinzip: (Last-In-First-Out-Speicher): Beim Auslesen eines Elementes wird das jeweils zuletzt gespeicherte Element zuerst ausgelesen, danach das vorletzte etc. Der Kellerspeicher wird daher auch als Stapel bezeichnet: Elemente werden sozusagen übereinander gestapelt und dann wieder in umgekehrter Reihenfolge vom Stapel genommen.

- Die Operation push packt ein Element des Parametertyps *T* auf den Stapel, pop entfernt das oberste Element.
- Mittels top kann man sich das oberste Element anschauen (ohne es zu entfernen).
- All diese Operationen sind als Funktionen mit einem (ersten) Parameter vom Typ Stack (genauer natürlich Stack(T)!) als Eingabe definiert.
- empty erzeugt einen leeren neuen Stapel, is_empty testet auf leeren Stack.

Der Stapelspeicher



[addp]

Abb. 1. Verdeutlichung des Stack-Prinzips

Der Stapelspeicher

Eine besondere Behandlung erfordert noch der Fall, dass versucht wird, ein Element vom leeren Stack zu lesen oder wenn es zu einem Überlauf des Stacks kommt.

- Dies kann über eine Ausnahme (Exception, Signal) des Typs StackException signalisiert werden, die beim Aufruf der Methoden pop, push und top auftreten kann.
- Der Datentyp der Datenelemente kann beliebig sein:
  - Numerische Werte
  - Zeichenketten
  - Datenstrukturen
- Statisch typisierte Programmiersprachen erfordern entweder ein virtuelles Template oder für jeden Datensatztyp eine eigene Implementierung. Dynamisch typisierte und teils auch funktionale Sprachen haben dieses Problem nicht.

Der Stapelspeicher

Algorithmik

Algorithmisch brauchen wir wieder eine Indexabbildungsfunktion, die jetzt aber parameterlos und zustandsbehaftet wird (d.h. zum Stapelobjekt gehört).

- Am einfachsten ist sowohl beim LIFO als noch beim folgenden FIFO Speicher die Einführung zweier Indexzeiger:
  - TOP: Zeigt auf das nächste freie Element in der lienaren Tabelle
  - BOTTOM: Zeigt auf das letzte belegte Element
- Man kann hier aber einfach nur einen Zeiger verwenden:
  - SP: Der Stackzeiger der auf das nächste freie Element zeigt und (SP-1) welches auf den obersten letzten Wert des Stapels verweist.

Der Stapelspeicher

```
function Stack(N) {
  return {SP:0,N:N,T:Array(N)}
}
function push(S,x) {
  if (full(S)) throw "Full";
  S.T[S.SP]=x;
  S.SP++;
}
function pop(S) {
  if (empty(S)) throw "Empty";
  x=S.T[S.SP-1];
  S.SP--;
  return x
}
function full(S) { return S.SP==S.N } function empty(S) { return S.SP==0 }
```

Alg. 1. Implementierung der Stackoperationen

Der Stapelspeicher

Die Komplexitätsklasse der Push und Pop Operationen liegt weiterhin bei konstanten O(1)!

Applikationen

1. Hardware: Fast jeder Mikroprozessor hat einen Hardware Stack (also in Register-Transfer Logik implementierten Stack)
   - Auf dem Stack werden temporäre variablen ablegt (für Berechnungen)
   - Auf dem Stack werden Funktionsrahmen beim Aufruf von von Funktionen abgelegt
2. Die Stack Maschine
   - Ein einfacher Prozessor mit 0-Operandenbefehlen
   - Alle Operationen laufen über einen (oder mehrere) Stacks
   - FORTH ist ein bekanntes Stackprozessor Systeme (Hardware+Software)
   - Aber auch WebAssembly (Browser) und Postscript (und PDF) Interpreter sind Stackmaschinen

Der Stackprozessor

1. Es gibt twei Stacks: Einen Daten- und einen Instruktions Stack *D* und *C*
2. Es gibt zwei Arten von Operationen:
   - Literale die eine Wert auf dem Stack ablegen (und C → D bewirken)
   - Unäre und binäre Arithmetische Operationen die ihre Operanden vom D-Stack holen und das Ergebnis auch dort wieder ablegen

Wir benötigen noch eine weitere Stack Operation *nth* die den n-obersten Werte (n>0) vom Stapel liest:

```
function nth(S,n) { return S.T[S.SP-n] } // n > 0
```

Jetzt müssen wir Befehle für unsere einfache Numerik VM definieren:

```
function LITERAL(x) { return x /*number*/ }
function ADD() { return '+' }
function END() { return '.' }
```

Auf dem Stack sind Daten und Code abgelegt. Code ist ein String, Daten ein numerischer Wert. That's all folks.

Wir brauchen nun einen Stackprozessor der das Programm vom Stack abarbeitet:

```
function run() {
  next=pop(C)
  while (next!='.') {
    switch (next) {
      case '+':
        a=pop(D); b=pop(D)
        push(D,a+b)
        break;
      default:
        if (typeof next=='number') push(D,next);
        else throw "EEXEC"
    }
    next=pop(C);
  }
}
```

Jetzt folgt der Stack und ein kleines Beispielsprogramm:

```
C=Stack(10)
D=Stack(10)

// Programm
push(C,END())
push(C,ADD())
push(C,LITERAL(3))
push(C,LITERAL(2))

run(C);
print(pop(D))
```

Jetzt wollen wir noch die Subtraktion hinzufügen. Bei der Addition ist die Reihenfolge der Operanden nicht relevant, bei der Subtraktion schon.

- Was sollen wir machen wenn die Operanden auf dem Stack in der falschen Reihenfolge liegen? Wir könnten eine *swap* Operation einführen!

```
function SUB() { return '-' }
function SWAP() { return '%' }
function run() {
  ...
    case '-': a=pop(D);b=pop(D);push(D,a-b); break;
    case '%': a=pop(D);b=pop(D);push(D,a);push(D,b); break;
  ...
```

Die Warteschlange

Um die Prinzipien der Spezifikation von parametrisierten Datenstrukturen zu vertiefen, betrachten wir kurz eine weitere Datenstruktur, nämlich die Warteschlange oder englisch Queue.

- Eine Queue realisiert das sogenannte FIFO-Prinzip, also einen First-In-First-Out-Speicher.
  - Das FIFO-Prinzip modelliert eine Warteschlange, bei der man sich hinten anstellt: »Wer zuerst kommt, mahlt zuerst«.
- Es gibt wieder zwei wesentliche Operationen, die zwar auf einer Tabelle operieren, aber ebenfalls wie beim Stack keine explizite Indizierung der Tabelle benörigen (daher auch kein RAM Modell):
  - *enter*: Schreibt einen neuen Wert in die Queue
  - *leave*: List einen alten Wert aus der Queue

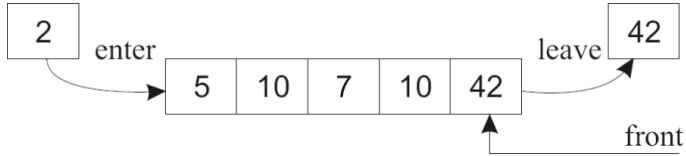
```
type Queue(T)
operators
  empty : → Queue
  new : n → Queue(n)
  enter : Queue × T → Queue
  leave : Queue → T
  bottom : Queue → T
  top : Queue → T
  is_empty : Queue → Bool
  is_full : Queue → Bool

axioms ∀q : Queue, ∀x, y : T
  leave (enter (empty, x)) = empty
  leave (enter (enter(q, x), y)) =
  enter (leave (enter(q, x)), y)
  top (enter (empty, x)) = x
  top (enter (enter(q, x), y)) =
  top (enter (q, x))
  is_empty (empty) = true
  is_empty (enter(q, x)) = false
  is_full (empty) = false
  is_full (enter(q, x)) = false
  is_full (n * enter(q, x)) = true
```

Def. 3. Die Spezifikation einer Queue sieht auf den ersten Blick sehr ähnlich zu der eines Stacks aus, obwohl das Verarbeitungsprinzip komplett entgegengesetzt ist

Die Warteschlange

Bezüglich Spezifikation: Im Unterschied zum Stack kann jetzt bei leave und bei front nicht der jeweilig letzte gespeicherte Parameter bearbeitet werden. Stattdessen ist es nötig, durch rekursiv aufgebaute Gleichungen sich bis zum jeweils »innersten« Wert des Queue-Terms vorzuarbeiten.



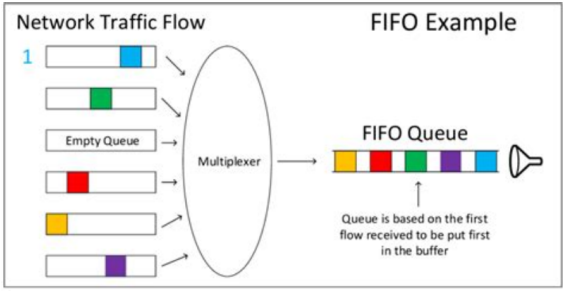
[addp]

Abb. 2. Verdeutlichung des Queue-Prinzips

Anwendungen

Warteschlangen werden ebenfalls in vielfältiger Weise in Softwaresystemen eingesetzt, da sie das Abarbeiten von Aufträgen in Eingangsreihenfolge realisieren können:

- so etwa zur Prozessverwaltung in Betriebssystemen (FIFO Scheduling),
- zur Speicherverwaltung,
- für Warteschlangen bei Druckaufträgen,
- für Nachrichtenpuffer in der Kommunikationssoftware etc.



[https://mungfali.com/explore/FIFO-Method-Diagram]

Abb. 3. Beispiel: Einsatz einer Queue im Nachrichtenmuliplexing (Multi-Queue System)

Algorithmik

- Das erste Element befindet sich immer auf der Position 0. Dies hat zur Folge, dass nach jedem Herausnehmen eines Elementes alle verbliebenen Elemente nach links verschoben werden müssten, was natürlich eine eher ungeeignete Variante darstellt.
- Wir brauchen jetzt tatsächlich zwei Zeiger um eine Queue mit einer linearen Tabelle zu implementieren:
  - HEAD: Zeigt auf das nächste freie Element (davor das jüngste)
  - BOTTOM: Zeigt auf das erste (älteste) Element in der Warteschlange
- Die zeiger werden bei enter und leave nur inkrementoiert:
  - enter: HEAD++
  - leave: BOTTOM++

Algorithmik

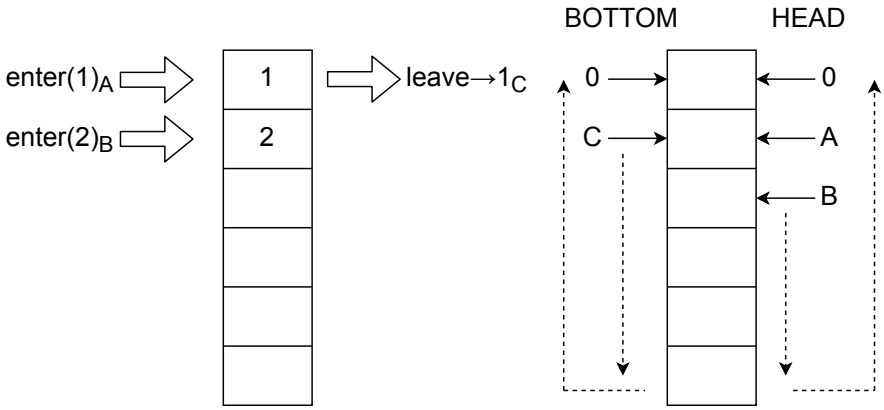


Abb. 4. Simulierter Ringpuffer mit zwei umlaufenden Zeigern. O: Start; A,B,C: Zustand nach enter/leave Operationen

Algorithmik

- Es wird ein zyklischer Ringpuffer durch die Modulo-Operation erzeugt: H%N, B%N mit N als Anzahl der Speicherzellen.
- Den Füllstand einer Queue über die beiden Zeigen zu bestimmen (empty, full) ist nicht eindeutig. Gleichheit von HEAD und BOTTOM liegt bei Leere und vollständiger Füllung vor! Daher wird ein Zähler verwendet (oder es dürfen nur N-1 Feldelemente befüllt werden).

```
function Queue(N)   { return { H:0,B:0,T:Array(N), N:N, n:0 } }
function enter(Q,x) { Q.T[Q.H]=x; Q.H=(Q.H+1) % Q.N; Q.n++; }
function leave(Q,x) { x=Q.T[Q.B]; Q.B=(Q.B+1) % Q.N; Q.n--; return x}
function empty(Q)   { return Q.n==0 }
function full(Q)    { return Q.n==Q.N }
function bottom(Q)  { return Q.n>0 && Q.T[Q.B] }
function top(Q)     { return Q.n>0 && Q.T[abs((Q.H-1)%Q.N)] }
```

Alg. 2. Queue Operationen

Algorithmik

Ein Problem der hier vorgestellten Implementierung ist die Größenbeschränkung des Feldes. So werden nur so lange Elemente eingefügt, bis die Kapazität des Feldes erschöpft ist. Ein Ausweg wäre das Erzeugen eines neuen, größeren Feldes und das Umkopieren der Elemente aus dem alten Feld. Dennoch haben Felder durchaus vorteilhafte Eigenschaften, wie etwa den günstigen Platzverbrauch (keine zusätzlichen »Verwaltungsinformationen«) oder die kompakte und dadurch Cache-freundliche Organisation der Daten.

- Die Zeitkomplexität ist konstant und beträgt O(1) für alle Operationen.
- Der Speicherplatzbedarf wird dagegen durch das Anlegen eines Feldes einer vorgegebenen Größe N verursacht und ist somit unabhängig von der tatsächlichen Anzahl der Elemente im Stack bzw. in der Queue. Für die Speicherplatzkomplexität kann demnach O(N) angegeben werden.

Hash Tabellen

Das Grundprinzip von Hashverfahren lässt sich mit wenigen Aussagen charakterisieren:

1. Die Speicherung der Datensätze erfolgt in einem Feld mit Indexwerten von 0 bis N− 1, wobei die einzelnen Positionen oft als »Buckets« bezeichnet werden.
2. Eine Hashfunktion h bestimmt für ein Element e die Position h(e) im Feld.
3. Die Funktion h sorgt für eine »gute«, kollisionsfreie bzw. kollisionsarme Verteilung der zu speichernden Elemente.

Da es in der Regel sehr viel mehr möglicherweise zu speichernde Elemente als die N Positionen im Feld gibt und die zu speichernden Elemente vorher unbekannt sind, kann die Funktion nicht allen möglichen Elementen unterschiedliche Positionen zuweisen.

Wird zwei Elementen dieselbe Position zugewiesen, spricht man von einer **Kollision**. Neben einer möglichst gut streuenden **Hashfunktion** ist die Behandlung von Kollisionen ein zentrales Merkmal von Hashverfahren.

Hash Tabellen

Parametrisierung eines Problems

1. Es gibt K mögliche Schlüssel (Schlüsselgröße)
   - Numerische Schlüssel, z.B. Matrikel- oder Personalnummer
   - Zeichenketten, z.B. Nachnamen
   - Quersummen
   - Zusammengesetzte Datenstrukturen
2. Es gibt eine maximale Anzahl von möglichen Datensätzen L (Problemgröße)
3. Es gibt eine feste Größe der Tablle mit N Feldern (Tabellengröße)
4. Es gibt eine Hashfunktion $h(k)$: $k \rightarrow i$, die einen Schlüssel auf einen Index der Tabelle abbildet, so dass $k \in \mathbb{K}$, $i \in [0,N\text{-}1]$.
5. Es gibt eine aktuelle Belegung mit $\alpha=m/N$, wobei m die Anzahl der gespeicherten Schlüssel ohne Überläufer ist.

Hash Funktionen

Man kann sich leicht klar machen, dass bei einem genügend großen Wertevorrat auch immer *n* Elemente gefunden werden können, die durch *h* auf dieselbe Position abgebildet werden – ein Hashverfahren kann entarten. Durch eine gute Hashfunktion kann man die Wahrscheinlichkeit einer Entartung verringern, aber nie ganz ausschließen – wohl ein Grund, weshalb viele Programmierer Hashverfahren etwas misstrauisch betrachten und sie trotz ihrer guten Eigenschaften relativ selten einsetzen.

- Betriebssysteme verwenden Hashtabellen für einen effiziente Verwaltung von Ressourcen

Komplexität

Einfügen, Löschen, Suche:

- Best case: O(1)
- Average case: O(1) ($\leq$ O(N))
- Worst case: O(N)
- Die Anzahl der zum Suchen, Einfügen oder Entfernen eines Schlüssels benötigten Schritte hängt im wesentlichen vom Belegungsfaktor α ab.

Hash Funktionen

- Von zentraler Bedeutung für Hashverfahren sind die Hashfunktionen.
- Die Hashfunktionen hängen natürlich vom Datentyp der zu speichernden Elemente und der konkreten Anwendung ab (die Mengen von zu speichernden string-Werten können je nach Anwendung sehr unterschiedlich aussehen).

Divisions-Rest-Methode

- Für Integerwerte als Suchschlüssel *k* wird oft als Hashfunktion direkt die Modulo Funktion verwendet:

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

Dies funktioniert in der Regel allerdings nur dann gut, wenn ***N*** eine (große) **Primzahl** ist, die nicht nahe an einer großen **Zweierpotenz** liegt, da sonst unbeabsichtigt besondere algebraische Eigenschaften der eingegebenen Schlüssel eine Gleichverteilung verhindern.

- Gerade künstlich erzeugte Suchschlüssel haben oft verborgene algebraische Eigenschaften, die nicht auf den ersten Blick zu erkennen sind, Z.B.
  - dass das gerne gewählte N = 2i etwa die Eigenschaft »ungerade« erhält,
  - werden beispielsweise generierte Artikelnummern durch eine der drei Kontrollziffern 1, 3 oder 7 abgeschlossen, würde ein Hashen mit N = 2i die Werte nur auf ungerade Adressen abbilden.

Für andere Datentypen kann eine Rückführung auf Integerwerte erfolgen:

- Bei Fließpunktzahlen kann man Mantisse und Exponent addieren.
- Bei Strings könnte man deren ASCII/Unicode-Wert oder den einiger Buchstaben davon, eventuell jeweils mit einem Faktor gewichtet, addieren. In einer konkreten Variante werden für einen String *s* der Länge *l* aus 1-Byte-Zeichen *l* (initial) zufällige Elemente $a_i \in \{0, \ldots, N-1\}$ gewählt und der Hashwert wie folgt berechnet:

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

- Hashfunktionen sollen die Werte natürlich besonders gut »streuen«. Daher ist es oft sinnvoll, eventuell von Besonderheiten der Eingabewerte abhängige Funktionen zu wählen (Buchstaben des Alphabets tauchen in Namen unterschiedlich häufig auf!). Allerdings müssen Hashfunktionen auf jeden Fall effizient berechenbar sein (konstanter Zeitbedarf, auf keinen Fall abhängig von der Anzahl der gespeicherten Werte!).

- Das Matrikelnummer-Beispiel zeigt, dass eine schlecht gewählte Hashfunktion die Güte des Verfahrens schnell negativ beeinflussen kann. Die Hashtabelle besteht hierbei aus 100 Buckets, gespeichert werden sollen Studentendatensätze, identifiziert durch die Matrikelnummer MATRNR. Die 6-stellige MATRNR wird fortlaufend vergeben.

  - Werden die ersten beiden Stellen von MATRNR als Hashwerte ausgewählt, werden die Datensätze eines Jahrgangs alle auf sehr wenige Positionen abgebildet.
  - Die letzten beiden Stellen von MATRNR hingegen verteilen die Datensätze gleichmäßig auf alle Positionen.

Die von der Hashfunktion zu gegebenen Schlüsseln gelieferten Hashadressen sollten also über dem Adreßbereich gleichverteilt sein, und zwar selbst dann, wenn die Schlüssel aus $\mathbb{K}$ alles andere als gleichverteilt sind (etwa bei der Vorliebe von Programmierern für Namen wie x, x1, x2, y1, y2, z1, z2).

Die multiplikative Methode

Der gegebene Schlüssel wird mit einer irrationalen Zahl multipliziert; der ganzzahlige Anteil des Resultats wird abgeschnitten. Auf diese Weise erhält man für verschiedene Schlüssel verschiedene Werte zwischen 0 und 1; für Schlüssel 1, 2, 3,.., *n* sind diese Werte ziemlich gleichmäßig im Intervall [0,1) verstreut

Von allen Zahlen Θ, 0 ≤ Θ ≤ 1, führt der goldene Schnitt zur gleichmäßigsten Verteilung:

$$\phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

Damit erhalten wir folgende Hashfunktion:

$$h(k) = \lfloor N\left(k\phi^{-1} - \lfloor k\phi^{-1} \rfloor\right) \rfloor$$

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

Man kann die Berechnung von $h(k)$ noch beschleunigen, wenn man ganze Zahlen im Rechner als Bruchzahlen mit Dezimalpunkt vor der höchstwertigen Ziffer ansieht, und wenn man für m eine Zweierpotenz wählt; dann läßt sich die Berechnung von $h(k)$ mit einer ganzzahligen Multiplikation und einer (oder zwei) Shift-Operation(en) vornehmen.

- Neben diesen beiden Methoden gibt es noch zahlreiche andere, die z.B.
  - nach einer Transformation des Schlüssels (in ein anderes Zahlensystem oder durch Quadrieren oder durch Falten auf kurze Länge mit Verknüpfen von Teilstücken) einzelne Ziffernpositionen auswählen.

Praxis

Perfektes und universelles Hashing

- Ist die Anzahl der zu speichernden Schlüssel nicht größer als die Anzahl der zur Verfügung stehenden Speicherplätze, gilt also für die Teilmenge $\boldsymbol{K}$ der Menge $\mathbb{K}$ aller möglichen Schlüssel $|\boldsymbol{K}| \leq N$, so ist eine kollisionsfreie Speicherung von $\boldsymbol{K}$ immer möglich!

- Wenn wir $\boldsymbol{K}$ kennen und $\boldsymbol{K}$ fest bleibt, können wir leicht eine injektive Ab-bildung $h : \boldsymbol{K} \rightarrow \{ 0,..,N-1\}$ z.B. wie folgt berechnen:

  - Wir ordnen die Schlüssel in $\boldsymbol{K}$ lexikographisch und bilden jeden Schlüssel auf seine Ordnungsnummer ab.
  - Wir haben damit eine perfekte Hashfunktion, die Kollisionen gänzlich vermeidet.
  - Eine solche Situation ($\boldsymbol{K}$ fest und vorher bekannt) liegt z.B. dann vor, wenn den Schlüsselworten einer Programmiersprache feste Plätze in einer Symboltabelle zugeordnet werden sollen.
  - Dieser Fall ist aber eher die Ausnahme als die Regel.
  - Im allgemeinen kennen wir $\boldsymbol{K} \subseteq \mathbb{K}$ nicht und können selbst dann, wenn $|\boldsymbol{K}| \leq N$ bleibt, nicht sicher sein, daß Kollisionen vermieden werden.

Perfektes und universelles Hashing

Lum, Yuen und Dodd hatten festgestellt, dass das Divisions-Rest-Verfahren im Durchschnitt die besten Resultate liefert!

- Bei der Analyse der Effizienz von Hashverfahren geht es uns in erster Linie um die durchschnittliche Laufzeit für die Operationen Suchen, Einfügen und Entfernen.
  - Im schlimmsten Fall sind diese Operationen extrem langsam; dieser Fall ist leicht direkt aus der Beschreibung der Verfahren ableitbar.
- Wir werden stets zwei Erwartungswerte C_{n+} und C_{n-} angeben, bezogen auf feste Tabellengröße m.
- Dabei ist C_{n+} der Erwartungswert für die Anzahl der betrachteten Einträge der Hashtabelle bei erfolgreicher Suche, C_{n-} der Erwartungswert für die Anzahl der betrachteten Einträge der Hashtabelle bei erfolgloser Suche.

Komplexität beim Hashtabellen

- Den Werten liegt eine Wahrscheinlichkeitsverteilung (der Schlüssel) zugrunde.
  - Dem Entfernen eines Datensatzes muß stets eine erfolgreiche Suche vorausgehen; entsprechend ist der Aufwand für das Entfernen gerade C_{n+} , wenn der betreffende Eintrag lediglich als entfernt markiert wird.
  - Dem Einfügen eines Datensatzes muß stets eine erfolglose Suche vorausgehen; entsprechend ist der Einfüge-Aufwand gerade C_{n-} , wenn der betreffende Datensatz einfach an der ersten gefundenen freien Stelle eingetragen wird.

Umgang mit Kollision und Überläufer

Folgende Verfahren können eingesetzt werden wenn eine Kollision festgestellt wird, d.h. der mit $h(k_1)$ berechnete Platz ist bereits mit einem Schlüssel k_2 mit $k_1 \neq k_2$ belegt, d.h. das einzufügende Element wird zum Überläufer

1. Verkettung der Überläufer mit Listen - man bringt die Überläufer außerhalb der Tabelle unter
2. Offene hashverfahren - man bringt die Überläufer in der Tabelle woanders unter:
   - Lineares Sondieren
   - Quadratisches Sondieren
   - Double Hashing
   - Ordered Hashing
   - Robin-Hood Hashing
   - Coealesced Hashing
3. Dynamische Hashverfahren - Hashverfahren für stark wachsende oder schrumpfende Datenbestände

AlgoSkript.pdf p 56 Algorithmen und Datenstrukturen, Reith

Algorithmen und Datenstrukturen (Thomas Ottmann, Peter Widmayer) (Z-Library).pdf p 169 Kap. 4 hashverfahren