Algorithmen und Datenstrukturen

2. Semester

Dozent

Prof. Dr. rer. nat. Peter Jüttner

peter.jüttner@th-deg.de

Mitschrift

Christoph Stephan

chm.stephan@outlook.com

Inhaltsverzeichnis

1 Begriff Algorithmus

Algorithmus: Definierte Berechnungsvorschrift, die aus einer Menge von Eingaben eine Menge von Ausgaben erzeugt.

1.1 Beispiele

- Kochrezepte «... man nehme ...»
- Bedienungsanleitungen
- Beipackzettel für Medikamente «Tablette in Wasser auflösen ...»
- Anleitung zum Zusammenbau von für IKEA Möbeln
- Anleitung zum Zusammenbau von LEGO Modellen (Besonderheit? Bilder)
- Computerprogramme «if a<b ... else ...»

1.2 Eigenschaften

- **Endlichkeit der Beschreibung** (Gegenbeispiel (Gegenbeispiel $1+1/2+1/4+1/8+1/16+...$)
- **Effektivität**, d.h. jeder Schritte des Algorithmus ist ausführbar
- **Terminierung**, d.h. der Algorithmus kommt immer(!) in endlicher Zeit zu einem Ende[1] (Gegenbeispiel: Berechnung der Zahl Pi)
- **Effizienz**, d.h. Verhältnis von Aufwand und Leistung, ein Algorithmus ist z.B. effizienter, wenn er mit weniger Rechenoperation auskommt oder mit weniger Speicherplatz als ein anderer[2] (Achtung: Effizienz $\neq$ Effektivität !)
- **Determinismus**[3], d.h. der Ablauf ist eindeutig vorgeschrieben (Gegenbeispiel «... man nehme ein Pfund Hackfleisch von Rind oder vom Schwein ...»)
- **Determiniertheit**[4], d.h. das Ergebnis des Algorithmus ist bei gleichen Eingabedaten immer gleich

Anmerkungen:

1. Determinismus und Determiniertheit sind nicht abhängig voneinander.
   D.h. ein nicht-deterministischer Algorithmus muss nicht automatisch nicht-determiniert sein.
   Beispiel: Finden des kleinsten Elements in einem Integer Array: Durchsuche das Array beginnend mit dem kleinsten Index oder durchsuche das Array beginnend mit dem größten Index.

[1] diese Eigenschaft ist nicht immer auf Anhieb zu erkennen und nicht allgemein zu beweisen!
[2] Effizienz ist «relativ», bei kleinem Speicherplatz ist ein langsamerer Algorithmus u.U. effizienter
[3] das zugehörige Adjektiv ist deterministisch
[4] das zugehörige Adjektiv ist determiniert

Beispiel: Nehmen Sie eine Zahl x ungleich 0; Addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x oder subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x

2. Die gängigen Programmiersprachen C, C++, Java, C# sind deterministisch
3. Nicht-deterministisch erscheinende Abläufe entstehen bei parallelen Abläufen von Software, z.B. in verschiedenen Prozessen auf einem Mikrocontroller

- **Semantik**, d.h. die Bedeutung des Algorithmus

Anmerkungen:

1. Die tatsächliche Semantik eines Algorithmus kann von der beabsichtigten Semantik abweichen. In diesem Fall ist der Algorithmus falsch!
2. Verschiedene Algorithmen können die gleiche Semantik haben.

- **Korrektheit**, d.h. die beabsichtigte Semantik entspricht der tatsächlichen Semantik des Algorithmus

Achtung: Die Korrektheit von Algorithmen ist nur sehr eingeschränkt beweisbar!

1.3 Bausteine von Algorithmen

- elementare Operationen, z.B.: $a + b, c * d, a < 5, >$
- sequentieller Ablauf, z.B.: a+b; c-d;
- paralleler Ablauf (Mehrprozessorsysteme!)
- bedingte Ausführung, z.B.: if a<b ...
- Schleifen, z.B.: for (i = 1 ...), while (h == 7) ...
- Unterprogramme, z.B.: a = f(b);
- Rekursion, z.B.: f(int i) if (...) else f(i-1)
- (Sprünge, z.B.: goto marke;)

Anmerkungen:

1. Die Konstrukte elementare Operationen, Sequenz, Bedingung, Schleifen (oder Sprünge) sind ausreichend, alle auf Rechnern programmierbaren Algorithmen zu beschreiben (auch andere Kombinationen möglich)
2. Es gibt Probleme, die sich nicht mittels eines Programms lösen lassen, z.B.: Terminierungsproblem

$\Rightarrow$ Theorie der Berechenbarkeit

1.3.1 Existenz nicht berechenbarer Funktionen

Vorbedingung: Jeder Algorithmus lässt sich in einem endlichen, fest definierten Alphabet beschreiben
(andernfalls ließe sich der Algorithmus nicht als Programm einer Programmiersprache darstellen)

- $A := \{a_1; \ldots; a_n\}$ ein Alphabet mit der Ordnung $a_1 < a_2 < \ldots < a_n$.

- $A*$:= Menge der Texte (Zeichenketten, endlichen Folgen, Worte), die aus A gebildet werden können
- $A* = \{«\_», «a_1», «a_2», \ldots, «a_1a_1», «a_1a_2», \ldots\}$
- Die Elemente von $A*$ können der Länge nach aufgelistet werden. Zu einer Länge l gibt es nl verschiedene Texte (endlich viele)
- Durch $«b_1b_2 \ldots b_k» < «c_1c_2 \ldots c_k»$
  $\Leftrightarrow b_1 < c_1$
  $\vee (b_1 = c_1 \wedge b_2 < c_2)$
  $\vee \ldots$
  $\vee b_k < c_k$
  wird eine Ordnung auf A* definiert.
- Aus der Tatsache, dass es nur endlich viele Texte einer Länge l gibt und dass über $A*$ eine Ordnung definiert werden kann, folgt $A*$ ist abzählbar (d.h. $A*$ kann durchnummeriert werden)
- Betrachten wir nun speziell einstellige Funktionen $f : \mathbb{Z} \to \mathbb{Z}$ ($\mathbb{Z}$ ganze Zahlen)
- Wie oben erläutert, gibt es nur abzählbar viele solcher Funktionen, die auch berechenbar sind.
- Es gibt aber insgesamt mehr Funktionen, wie die folgende Überlegung zeigt:
  - Wir betrachten die Menge $F := \{f : \mathbb{Z} \to \mathbb{Z}\}$ der einstelligen Funktionen auf $\mathbb{Z}$ und nehmen an, dass diese Menge ebenfalls abzählbar ist. $\Rightarrow$ jedes f aus F hat eine Nummer i, d.h. $F = \{f_1, f_2, \ldots\}$
  - Sei nun $g : \mathbb{Z} \to \mathbb{Z}$ definiert durch $g(x) = f_{abs(x)}(x) + 1$
  - $\Rightarrow$ es gilt für $i = 1, 2, \ldots$ $g(i) \neq f_i(i)$
  - $\Rightarrow$ für $i = 1, 2, \ldots$ gilt immer $g \neq f_i$
  - $\Rightarrow$ g kommt in $\{f_1, f_2, \ldots\}$ nicht vor, ist aber eine einstellige Funktion auf $\mathbb{Z}$ und müsste somit in F vorkommen $\Rightarrow$ Widerspruch
  - $\Rightarrow$ Der Widerspruch lässt sich nur auflösen, wenn die Annahme fallengelassen wird, F sei abzählbar.
  - $\Rightarrow$ Es gibt nicht berechenbare Funktionen[5]

Beispiele nicht berechenbarer Funktionen: [6]
- Halteproblem: Terminiert ein Algorithmus x mit Eingabe y
- Erreicht ein Algorithmus eine bestimmte Stelle
- Sind zwei Algorithmen gleich (d.h. haben sie immer das gleiche Ergebnis)?
- Ist ein Algorithmus korrekt?

1.4 Beweis der Terminierung von Algorithmen

Die Terminierung eines Algorithmus lässt sich allgemein nicht beweisen. Für bestimmte Algorithmen ist dies aber möglich.

[5] diese Tatsache war schon in den 30er Jahren des vorigen Jahrhunderts bekannt
[6] für einen einzelnen gegebenen Algorithmus lässt sich das u.U. entscheiden, nicht aber allgemein

2 Komplexität von Algorithmen

Ziele:

1. für die Lösung eines Problems mittels eines Algorithmus: Korrektheit des Algorithmus.
2. Problemlösung durch Algorithmus mit geringem oder geringstem Aufwand[1]

2. Ziel u.U. genau so wichtig wie 1., speziell im harten Echtzeitbereich, z.B.:

- Airbag, ABS, ESP (lebenswichtig)
- Motorsteuerung (gesetzliche Vorgaben)
- Fernbedienung, Navigation (Komfort)

2.1 Komplexitätstheorie für Algorithmen

⇒ Schätzen des Aufwands eines konkreten Algorithmus
⇒ Angabe eines Mindestaufwands für die Lösung eines bestimmten Problems oder einer Klasse von Problemen.

2.1.1 Komplexitätsklassen

- für Algorithmen zur Lösung eines Problems
- für Probleme, die mittels eines Algorithmus gelöst werden sollen

Algorithmus a löst Problem p

- Komplexitätsklasse (Algorithmus a) > Komplexitätsklasse (Problems p)
  ⇒ besseren Algorithmus suchen
- Komplexitätsklasse (Algorithmus a) = Komplexitätsklasse (Problems p)
  ⇒ nur noch Feintuning möglich

Problem abhängig von einem (oder mehreren) Größenparametern (z.B.: Feld der Länge n, Baum der Tiefe t, Struktur mit x Bytes)
⇒ wie wächst die Komplexität der Lösung des Problems mit dem Größenparameter
⇒ die Komplexität der Lösungsalgorithmen wächst ebenfalls mit dem Größenparameter

Möglichst unabhängig von einem konkreten Rechner/HW/Controller ⇒ Berechnung der Komplexität in Rechenschritten (für die Laufzeit)

- wie häufig wird eine Operation ausgeführt

[1] mit Aufwand kann Zeit und/oder auch Platzbedarf gemeint sein

- wie laufzeitintensiv ist eine Operation im Vergleich zu anderen Operationen (ggf. ausschließliche Betrachtung laufzeitintensiver Operationen)

⇒ maschinenunabhängiges Maß für die Rechenzeit

Allgemeine Form: $f : \mathbb{N} \to \mathbb{N}$ ($\mathbb{N}$ hier für die natürlichen Zahlen)
wobei $f(n) = a$ steht für: «Ein konkretes Problem der Größe n erfordert Aufwand a.»

- Die «Problemgröße» n bezeichnet dabei meist ein grobes Maß für den Umfang einer Eingabe, z.B.:
  - die Anzahl der Elemente einer Liste
  - Anzahl Elemente in einem Feld
  - Größe eines bestimmten Eingabewertes
- Der «Aufwand» a ist in der Regel ein grobes Maß für die Rechenzeit (und ggf. den Speicherplatz)
- Die Rechenzeit zählt die Anzahl der Rechenoperationen und ggf. Anzahl der Speicherzugriffe,
  ⇒ Maschinenunabhängiges Maß

2.1.2 Komplexitätsklassen «O-Notation»

- Abschätzungen mit vereinfachenden Annahmen
- obere Schranke für die Komplexität
- abhängig vom Größenparameter n ($n \in \mathbb{N}$) des Problems

⇒ Komplexität Problem $p(n) <= O(g(n))$, d.h. es gibt eine Funktion $g(n)$ in den natürlichen Zahlen für die gilt: die Lösung des Problems p (abhängig vom Größenparameter n) erfordert mindestens $g(n)$ Schritte

Mathematische Definition: $f(n) \in O(g(n)) \Leftrightarrow \exists c, n_0 : \forall n \geq n_0 : f(n) \leq c * g(n)$

«f wächst nicht stärker als g» «f ist beschränkt durch g» «$f(n)$ hat höchstens die Komplexität $g(n)$»

Anwendung bei der Analyse von Algorithmen: Aufwandsfunktionen $f : \mathbb{N} \to \mathbb{N}$ wird durch Angabe einer einfachen Vergleichsfunktion $g : \mathbb{N} \to \mathbb{N}$ abgeschätzt.

Auflistung der Komplexität in steigender Form:

- $O(1)$ ⇒ konstanter Aufwand
- $O(\log n)$ ⇒ logarithmischer Aufwand
- $O(n)$ ⇒ linearer Aufwand
- $O(n \cdot logn)$
- $O(n^2)$ ⇒ quadratischer Aufwand
- $O(n^k)$ ⇒ polynomialer Aufwand ($k > 2$)
- $O(2^n)$ ⇒ exponentieller Aufwand

Anmerkungen:

- Summanden werden weggelassen, d.h. $O(n+5) = O(n)$
- Faktoren werden weggelassen, d.h. $O(5 \cdot n) = O(n)$
- Basen von Logarithmen werden (meist) weggelassen, d.h. $O(\log_2 n) = O(\ln n) = O(\log_{10} n)$

Beispiele:

- Suchen mittels Hashverfahren[2] $\Rightarrow O(1)$
- binäres Suchen in einem sortierten Array $\Rightarrow O(\log n)$
- lineares Suchen in einem unsortierten Array $\Rightarrow O(n)$
- Syntaktische Analyse von Programmen (Compiler) $\Rightarrow O(n)$
- Multiplikation Matrix-Vektor $\Rightarrow O(n^2)$
- Matrizen-Multiplikation $\Rightarrow O(n^3)$

[2] unter bestimmten Randbedingungen

3 Rekursion

Idee: Löse ein Problem dadurch, dass durch eine leicht durchzuführende Aktion das Problem auf einen einfacheren Fall des gleichen Problems zurückgeführt werden kann.

Beispiel: Suchen einer bestimmten Stelle (z.B.: Foto) in einem Buch durch Blättern.
→ **Einfache Aktion:** Blättern um eine Seite, nachschauen, ob das Foto gefunden ist. Falls nein: Suchen im restlichen Buch.

- «etwas auf sich selbst zurückführen»
- im Sinn der Programmierung «zurückführen auf einen einfacheren Fall des selben Problems»
- mathematische Definitionen
  - 1 ist eine natürliche Zahl
  - ist n eine natürliche Zahl, dann ist auch $n + 1$ eine natürliche Zahl
- rekursiver Algorithmus
- rekursive Datenstruktur
- erlaubt eine unendliche Menge von Objekten mit einer endlichen Definition zu beschreiben
- erlaubt eine unendliche Anzahl von Berechnungen durch eine endliche Definition zu beschreiben
- rekursive Algorithmen beschreiben die Lösung «rekursiver» Probleme auf einfache Art[1]
- rekursive Algorithmen arbeiten auf rekursiven Datenstrukturen auf einfache Art[2]
- Im Sinn der Programmiersprache bedeutet Rekursion, dass eine Funktion sich direkt oder indirekt selbst aufruft.
- **direkte Rekursion:** Funktion ruft sich selbst im eigenen Funktionsrumpf auf.
- **indirekte Rekusion:** Funktion f ruft eine Funktion g auf, die direkt oder indirekt wiederum f aufruft
- **kaskadenartige Rekusion:** In der Funktion f werden mindestens zwei Aufrufe von f «nebeneinander» aufgerufen.
  «nebeneinander» bedeutet hier dass die rekursiven Aufrufe verknüpft sind, z.B.: durch einen Operator oder Parameter eines anderen Funktionsaufrufs sind. Der äußere Aufruf kann erst beendet werden, wenn die kaskadenartigen rekursiven Aufrufe beendet sind.
- **geschachtelte Rekusion:** In der Funktion f wird f in der Form $f(..., f(...), ...)$ aufgerufen.
- Rekursion benötigt immer eine Terminierung, d.h. einen Fall, der zu keinen weiteren rekursiven Aufrufen führt (z.B.: bei Fakultät i==0)
- Mittels Rekursion lassen sich viele Aufgabenstellungen «einfach» lösen, falls die Aufgabe selbst rekursiv lösbar ist.

[1] der Algorithmus ist dabei nicht immer der effizienteste oder im Extremfall nicht effektiv
[2] der Algorithmus ist dabei nicht immer der effizienteste oder im Extremfall nicht effektiv

3.1 Behandlung von rekursiven Funktionsaufrufen

- jeder rekursive Aufruf besitzt seine eigenen Parameter, auch wenn diese den selben Namen wie in der aufrufenden Funktion haben
- jeder rekursive Aufruf besitzt seine eigenen lokalen Variablen (Ausnahme static)
- Parameter und lokale Variable bleiben so lange gültig, bis der rekursive Aufruf beendet ist

3.2 Einsatzgebiete der Rekursion

Gut: Umgebungen, in denen ausreichend Speicherplatz auf dem Stack zur Verfügung steht
⇒ PC Umgebungen, «sehr große» eingebettete Systeme (Multimediasysteme im Kfz)

Schlecht: Systeme mit «wenig» Speicherplatz, z.B.: Mikrocontroller im Kfz

3.3 Schlußbemerkung

Rekursive Funktionen können in äquivalente nicht-rekursive Programme übergeführt werden («Entrekursivierung»).
⇒ einfach bei einfacher Rekursion ⇒ komplex bei kaskadenartiger Rekursion (erfordert in der Regel einen eigenen Stack zur Verwaltung)

4 Komplexe Datenstrukturen

- Stack (Stapel, Keller)
- Listen
- Queue (Schlange)

4.1 Stack (Stapel, Keller)

Zugriff nur «von oben»

- Ablegen oben
- Wegnehmen oben
- kein direkter Zugriff auf Elemente, die unterhalb des obersten Elements liegen

4.1.1 Definition

Ein Stack (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Elementen (vom Typ T), die nur an einem Ende der Folge gelesen oder beschrieben werden kann.

Funktionen

```
- stack* Push (stack *s, T e) /* fügt oben ein Element an */
- stack* Pop (stack *s) /* löscht oberstes Element */
  /* Voraussetzung: Stack ist nicht leer! */
- T Top (stack *s) /* liefert oberstes Element */
  /* Stack bleibt unverändert */
  /* Voraussetzung: Stack ist nicht leer! */
- bool Isempty (stack *s) /* Stack leer ? */
  /* Stack bleibt unverändert */
- stack* emptystack() /* Erzeugt leeren Stack */
```

Die Datenstruktur Stack kann durch ihre Eigenschaften formal vollständig beschrieben werden (Axiomatische Beschreibung)

Eigenschaften (Axiome):

- `isempty (emptystack()) = true`
- `not isempty(s)` $\Rightarrow$ `push(pop(s),top(s)) = s`
- `isempty(push(s,e)) = false`

- `top(push(s,e)) = e`
- `top(emptystack())` ⇒ Error
- `pop(emptystack())` ⇒ Error

4.1.2 Anwendung/Nutzen

- Speicherverwaltung bei Laufzeitsystemen:
  - Parameterübergabe
  - lokale/globale Variable
- Verwaltung der Aufrufinformation bei geschachtelten und rekursiven Funktionen
- Umwandlung von rekursiven Programmen in nicht-rekursive
- LIFO-Speicher (Last-In-First-Out)

4.1.3 Implementierung

Zum Beispiel als Array (Stapel wird auf Array gekippt)

4.2 Queue (Schlange)

Zugriff «an beiden Enden»
⇒ Anfügen nur hinten an der Schlange
⇒ Wegnehmen nur vorne
⇒ kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen

4.2.1 Definition

Eine Queue (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Elementen (vom Typ T), die nur an einem Ende («vorne») gelesen bzw. gelöscht werden kann und am anderen Ende («hinten») ergänzt.

Funktionen

```
- queue* Append (queue *q, T e) /* fügt hinten ein Element an */
- queue* Rest (queue *q) /* löschte erstes Element */
  /* Voraussetzung: Queue ist nicht leer! */
- T Top (queue *q) /* liefert erstes Element */
  /* Queue bleibt unverändert */
  /* Voraussetzung: Queue ist nicht leer! */
- Bool Isempty (queue * q) /* Queue leer ? */
  /* Queue bleibt unverändert */
- queue* emptyqueue() /* Erzeugt eine leere Queue */
```

Die Datenstruktur Queue kann (wie Stack) durch ihre Eigenschaften formal vollständig beschrieben werden (Axiomatische Beschreibung)

Eigenschaften (Axiome):

- `isempty (emptyqueue()) = true`
- `rest(append(q,e)) = emptyqueue()`, falls `isempty(q)`, sonst `append(rest(q),e)`
- `top(append(q,e) = e`, falls `isempty(q)`, sonst `top(q)`
- `isempty(append(q,e)) = false`
- `top(emptyqueue())` ⇒ Error
- `rest(emptyqueue())` ⇒ Error

4.2.2 Anwendung/Nutzen

- Verwaltung von Betriebsmitteln (z.B.: Drucker, Prozessor) in Betriebsystemen:
  - Prozesse, die auf ein Betriebsmittel warten werden in eine Warteschlange eingehängt
  - Verschiedene Warteschlangen, z.B.: individuell für jedes Betriebsmittel oder auch prioritätsgesteuert.
- FIFO (First-in-First-Out) Speicher

4.3 Einfach Verkettete Liste (List)

Zugriff nur «am hinteren Ende»
⇒ Anfügen und Wegnehmen hinten an der Liste
⇒ kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen

4.3.1 Definition

Eine einfach verkettete Liste (über einem bereits existierenden Datentyp T) besteht aus einer Folge von Listenelementen, die wiederum aus einem Element vom Typ T und aus einem Zeiger auf ein Listenelement bestehen. Der direkte (lesende und schreibende) Zugriff erfolgt am Ende der Liste[1].

- Im Unterschied zu den vorher besprochenen Datentypen Stack und Queue legen Listentypen eine bestimmte Implementierung (Struktur) über Pointer nahe.
- «Aufweichen» der rein axiomatische Definition des Datentyps über seine Eigenschaften
- Trotzdem kann auch eine Liste «rein axiomatisch» definiert werden.
- Im Unterschied zu Stack und Queue (die mittels Arrays implementiert werden) sind Listen als dynamischer Datentyp (theoretisch) nicht begrenzt.

Funktionen

```
- list* Tail (list *l) /* löschte erstes Element */
  /* Voraussetzung: Liste ist nicht leer! */
```

[1] Der Zugriff erfolgt wie bei einem Stack, daher lassen sich mittels Listen Stacks einfach implementieren. Mittels eines Stacks lässt sich auch eine Liste darstellen (eher ungewöhnlich)

```
- T Head (list * l) /* liefert erstes Element */
  /* Liste bleibt unverändert */
  /* Voraussetzung: Liste ist nicht leer! */
- list* Append (list *l, T e) /* fügt Element vorne an */
- bool Isempty (list *l) /* Liste leer ? */
  /* Liste bleibt unverändert */
- list* emptylist() /* Erzeugt eine leere Liste */
```

Die Datenstruktur Liste kann (wie Stack) durch ihre Eigenschaften formal vollständig beschrieben werden (Axiomatische Beschreibung)
Die Beschreibung ergibt sich analog zu den Axiomen von Stack.

4.3.2 Anwendung/Nutzen

Implementierung von Stacks und Queues

Vorteile: Dynamische Speicherverwaltung ermöglicht optimale Ausnutzung des Speichers, (theoretisch) keine Begrenzung der Größe

Nachteil: Zusätzlicher Speicherplatzverbrauch durch Verkettung über Pointer

Achtung: Speicherverwaltung muss sorgfältig durchgeführt werden (Memory Leaks!)[2]

4.4 Doppelt verkettete Liste (List)

Zugriff «am hinteren» und am «vorderen Ende»
⇒ Anfügen und Wegnehmen an beiden Enden
⇒ kein direkter Zugriff auf Elemente, die zwischen dem ersten und letzten Element liegen

- Spezieller Listenkopf zeigt auf den Anfang und das Ende der Liste.
- In jedem Listenelement zeigt ein Pointer auf das nächste Element (oder NULL) und auf das vorhergehende (oder NULL)
- vereinfacht den Zugriff
- ermöglicht die Implementierung von zweiköpfigen Queues

[2] dies gilt für alle dynamischen Datentypen, die über Pointer Speicherplatz selbst verwalten!

4.5 Bäume (Tree)

4.5.1 Definition

Ein Baum besteht aus einer endlichen Menge von Knoten K und einer endlichen Menge von gerichteten Kanten P (dargestellt als Pfeil) zwischen Knoten aus K. Es gibt maximal eine Kante von einem Knoten zu einem anderen.

Ein Baum hat einen ausgezeichneten Knoten, die Wurzel, die nicht Endknoten einer Kante ist.

Gibt es eine Kante von einem Knoten k_1 zu einem Knoten k_2, dann ist k_1 der Elternknoten von k_2. k_2 ist Kind von k_1. Ein Knoten k_2 ist erreichbar von einem Knoten k_1, wenn es eine Folge von Knoten $k_1, k_x, \ldots, k_{x+n}, k_2$ gibt, so dass k_1 Elternknoten von k_x ist, k_{x+i} Elternknoten von k_{x+i+1} für alle i von 0 bis $(n-1)$ und k_{x+n} ist Elternknoten von k_2.

Ist k_1 Elternknoten von k_2, so ist k_2 von k_1 aus auch erreichbar. Knoten, die keine Kinder haben, heißen Blatt.

Ist ein Knoten l erreichbar von einem Knoten k, so ist k Vorgänger von l und l Nachfolger von k.
Jeder Knoten ist auf nur genau eine Weise von der Wurzel aus erreichbar.

Der Grad $g(k)$ eines Knotens k ist die Anzahl seiner Kinder.

Die Tiefe (oder Höhe) $t(k)$ eines Knotens k ist definiert als
0, falls k die Wurzel des Baums ist
$1 + t(k*)$, wenn $k*$ Elternknoten von k ist.

Ein Baum heißt geordnet, wenn die Reihenfolge der Verzweigungen in einem Knoten festgelegt ist. Ist dies nicht der Fall heißt der Baum ungeordnet.

4.5.2 Anwendung/Nutzen

- Darstellung von Unternehmensstrukturen
- Darstellung von Inhaltsverzeichnissen in Kapitel, Unterkapitel, Abschnitte
- Darstellung von statischen Programmstrukturen (Funktionen, Blöcke, Anweisungen) $\Rightarrow$ Compilerbau
- Darstellung mathematischer Ausdrücke, z.B.: $(a + b) * (c + d + e)$
- Darstellung von Codes
- Stammbäume
- Suchen
- Sortieren

4.5.3 Binär Bäume

Eigenschaften (Satz 1): Sei T ein nichtleerer binärer Baum mit Höhe h. Dann gilt:

1. Für $0 \leq i \leq h$ gilt, dass T maximal 2^i Knoten der Tiefe i besitzt.
2. T besitzt minimal $h+1$ und maximal $2^{h+1}-1$ Knoten.
3. Für die Anzahl n der Knoten in T gilt $\log(n+1)-1 \leq h \leq n-1$.

Definition

Ein geordneter Baum heißt binär, wenn er leer ist oder der Grad aller seiner Knoten kleiner oder gleich 2 ist.

Ein nichtleerer binärer Baum mit Höhe h heißt voll, wenn er $2^{h+1}-1$ Knoten besitzt.

Sei T_1 ein binärer Baum der Höhe h mit $n > 0$ Knoten. T_2 sei ein voller binärer Baum ebenfalls mit Höhe h. In T_2 seien die Knoten stufenweise von links nach rechts durchnummeriert (siehe vorheriges Beispiel). T_1 heißt dann vollständig, wenn er aus T_2 durch Wegstreichen der Knoten der Nummern $n+1, n+2, \ldots, 2^{h+1}-1$ erzeugt werden kann.

Sei T_1 ein (binärer) Baum. T_1 heißt ausgeglichen, wenn in jedem Knoten die Höhen der Teilbäume sich um höchstens 1 voneinander unterscheiden[3].

Funktionen

```
- b_tree* emptytree () /* erzeugt leeren Baum */
- b_tree* b_tree_new (b_tree *left, T e, b_tree *right)
  /* erzeugt aus zwei Teilbäumen und einem Knotenelement einen neuen Baum */
- b_tree* left (b_tree *t) /* gibt den linken Teilbaum zurück */
- b_tree* right (b_tree *t) /* gibt den rechten Teilbaum zurück */

- T element (b_tree *t) /* gibt den Inhalt der Wurzel zurück */
- bool ismepty (b_tree *t) /* Baum leer ? */
```

4.5.4 Weiter Baumtypen

- n-näre Bäume, d.h. Bäume, die in jedem Knoten n-fach verzweigen (unärer Bäum = verkettete Liste)
- beblätterte Bäume, d.h Bäume, die Information nur an den Blättern tragen. Die Knoten, die keine Blätter sind, haben nur Verzweigungen
- ...

4.5.5 Anwendungen von Bäumen

- Filesystem auf einem Rechner
- Teileliste einer Maschine
- Suchbäume (binäre Bäume)

[3] diese Bäume werden auch AVL-Bäume bezeichnet nach ihren Erfindern Adelson-Velski und Landis

4.5.6 Anwendungen binärer Bäume

⇒ Suchen, Sortieren
Betrachtet werden Bäume über einem Typ T mit einer linearen Ordnung $\leq$, d.h. für alle $a, b, c \in T$ gilt:

- $a \leq a$ (reflexiv)
- aus $a \leq b$ und $b \leq c$ folgt $a \leq c$ (transitiv)
- aus $a \leq b$ und $b \leq a$ folgt $a = b$ (antisymmetrisch)
- es gilt immer $a \leq b$ oder $b \leq a$ (totale Ordnung)

Sei $a_1, a_2, \ldots, a_n$ eine Folge von n Elementen aus T, so soll durch Sortieren die Folge so neu geordnet werden, dass gilt:
$a_i \leq a_j \leq a_k \leq \ldots \leq a_l$, wobei die Menge $\{a_1, a_2, \ldots, a_n\}$ der Menge $\{a_i, a_j, a_k, \ldots, a_l\}$ entspricht.

Ein binärer Baum ist geordnet, wenn

- er entweder leer ist oder
- alle Knoten des linken geordneten Teilbaums kleiner oder gleich sind als die Wurzel und
- alle Knoten des rechten geordneten Teilbaums größer oder gleich sind als die Wurzel

Ein geordneter binärer Baum wird als Suchbaum bezeichnet.

Operationen auf Suchbäumen:

- Suchen eines Elements x in einen Baum b
- Einfügen eines neuen Elements in den Baum
- Löschen eines Elements aus dem Baum
- Zusammenfügen von zwei Bäumen
- Bereinigen «ungünstiger» Suchbäume

Bereinigung nach jeder Operation, die den Baum verändert ist aufwändig
⇒ Nutzung von AVL-Bäumen
⇒ Nutzung von B-Bäumen

4.5.7 AVL Bäume (Suchbäume)

- benannt nach den russischen Mathematikern AdelsonVelskii und Landis (1962)
- **Kriterium:** für jeden (inneren) Knoten gilt: Höhe des linken und rechten Teilbaums differieren maximal um 1

Operationen auf AVL Bäumen:

- Einfügen, Löschen, Suchen wie zuvor
- Ändernde Operationen können die AVL-Eigenschaft zerstören
  ⇒ Reparieren mittels

Eine Rotation oder Doppelrotation beim Einfügen
Eine oder mehrere (Doppel-)Rotationen beim Löschen
- Operationen auf AVL-Bäumen erfordern $O(\log n)$ Aufwand

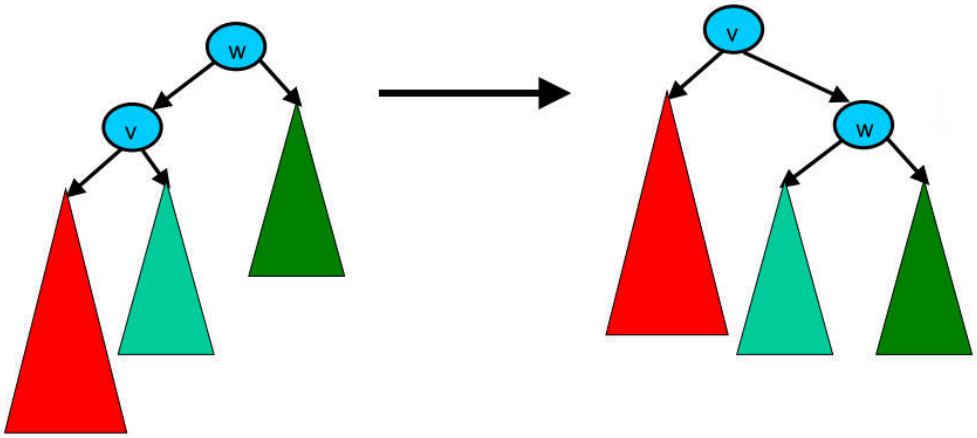

Abbildung 4.1 – Rotation

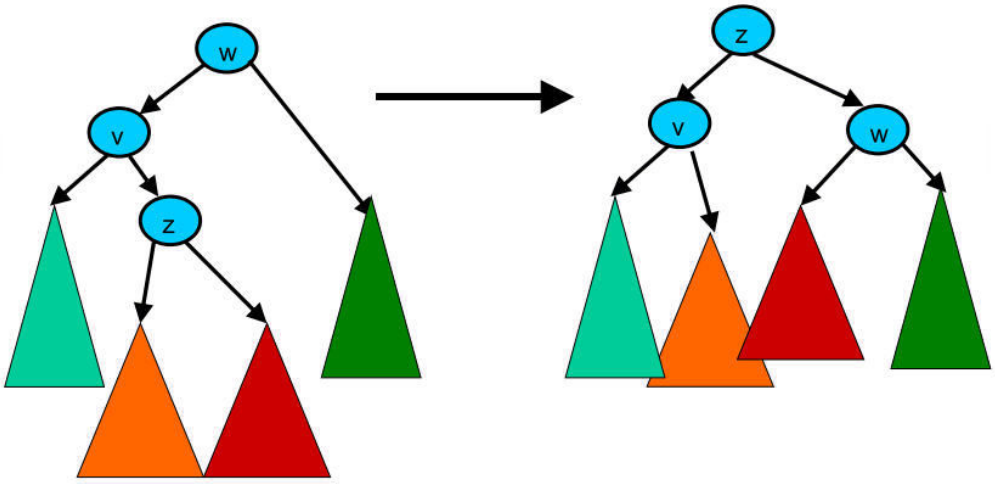

Abbildung 4.2 – Doppelrotation

Anmerkung Rotationen. Bei den Rotationen ändert sich die «horizontale Position» eines Knotens nicht, nur die vertikale Position wird verändert.

Zusammenfassung Rotationen auf oberstem Niveau:

1. Einfügen in linken Teilbaum des linken Kindes
2. Einfügen in rechten Teilbaum des linken Kindes
3. Einfügen in linken Teilbaum des rechten Kindes
4. Einfügen in rechten Teilbaum des rechten Kindes

Anmerkung 1. und 4. sind symmetrisch, 2. und 3. sind symmetrisch

Sonderfälle: Rotation von Teilbäumen

- Einfügen in linken Teilbaum des linken Kindes
  ⇒ Rotation mit linkem Kind
- Einfügen in rechten Teilbaum des linken Kindes
  ⇒ Doppelrotation mit linkem Kind
- Einfügen in linken Teilbaum des rechten Kindes
  ⇒ Doppelrotation mit rechtem Kind
- Einfügen in rechten Teilbaum des rechten Kindes
  ⇒ Rotation mit rechtem Kind

4.5.8 B-Bäume (Suchbäume)

- benannt 1978 nach ihrem Erfinder R. Bayer (B kann auch stehen für balanciert, breit, buschig, aber NICHT für binär)
- B-Bäume sind dynamische balancierte Mehrwegsuchbäume (d.h. nicht binär)

Vollständig ausgeglichener Mehrwegbaum:

- alle Wege von der Wurzel bis zu den Blättern gleich lang
- jeder Knoten gleich viele Einträge

⇒ Kriterium nur mit sehr hohem Aufwand einzuhalten
⇒ Modifikation zu B-Bäumen

Definition

- Jede Seite (= Knoten) enthält höchstens $2m$ Elemente.
- Jede Seite, außer der Wurzel, enthält mindestens m Elemente.
- Jede Seite ist entweder ein Blatt ohne Nachfolger oder hat $i + 1$ Nachfolger, wobei i die Anzahl ihrer Elemente ist.
- Alle Blattseiten liegen auf der gleichen Stufe, d.h. die Höhe vom Blatt zur Wurzel ist für alle Blätter gleich.

5 Komplexe Algorithmen

5.1 Sortierverfahren

- Sortieren bedeutet allgemein der Prozess des Anordnens einer gegebenen Menge von Daten in einer bestimmten Ordnung.
- Sortiert wird, um zu einem späteren Zeitpunkt schnell nach einem bestimmten Element der Menge suchen zu können.
- Beispiele sortierter Datenmengen: Telefonbücher, Indexe, Wörterbücher

Sortierverfahren:

- gibt es in zahlreichen Variationen
- zeigen (exemplarisch) wie ein Problem auf vielfältige Weise gelöst werden kann
- bieten ein gutes Beispiel, die Leistung verschiedener Algorithmen miteinander zu vergleichen

Ordnung (Definition): Gegeben sei ein Array eines Types T der Länge n, $n > 0$. Das Array ist bzgl. einer Ordnungsfunktion $f(T)$ geordnet, wenn gilt:
$f(\text{Array}[0]) \leq f(\text{Array}[1]) \leq \ldots \leq f(\text{Array}[n-1])$

5.1.1 Sortierverfahren für Arrays

Allgemeines

Ausgangspunkt: Array der Länge n, das bzgl. einer Ordnung $\leq$ zu sortieren ist.

Effizienz eines Sortieralgorithmus:

- Anzahl der notwendigen Vergleiche
- Anzahl der notwendigen Bewegungen (Austausch von Arrayelementen)

$\Rightarrow$ als Funktion von n (Anzahl der Elemente des Arrays)

Einfache Verfahren:

- Sortieren durch Einfügen
- Sortieren durch Auswählen
- Sortieren durch Austausch

Sortieren durch Einfügen:

- Durchlaufe das Array elementweise, beginnend mit dem 2. bis zum n-ten Element.
- Betrachte dabei jedes Element und sortiere es in die Sequenz der schon zuvor betrachteten Elemente ein. Dabei wird das einzusortierende Element mit den Vorgängern verglichen so lange die Vorgänger (Elemente mit kleinerem Arrayindex) größer sind oder der Anfang des Arrays erreicht ist.
- Das Ende des kompletten Sortiervorgangs ist erreicht, wenn das letzte Element des Arrays betrachtet und ggf. einsortiert wurde.

Sortieren durch binäres Einfügen: Verbesserung des direkten Einfügens durch Nutzen der Tatsache, dass die Sequenz, in die das gerade betrachtete Element eingefügt wird, bereits sortiert ist.

Gesamtaufwand wächst mit n^2

Sortieren durch direktes Auswählen: Prinzip:
⇒ Auswahl des kleinsten Elements im noch nicht sortierten Teil des Arrays
⇒ Austausch mit dem ersten Element des noch nicht sortierten Teilarrays
⇒ Analog mit dem Rest des Arrays

Mittel $n * (\ln n + g)$ **(g Eulersche Konstante)**

⇒ das Verfahren der direkten Auswahl ist günstiger als Einfügeverfahren

Sortieren durch direktes Austauschen (Bubblesort): Prinzip:
⇒ Mehrfaches Durchlaufen des Arrays (wie bisher)
⇒ Fortgesetztes Austauschen nebeneinander liegender Elemente im Array falls das hintere Element kleiner als das vordere ist.

Anzahl der Vergleiche: $(n^2 - n)/2$
Anzahl der Bewegungen: im Durchschnitt $3 * (n^2 - n)/4$

⇒ Sortieren durch direktes Austauschen ist den vorherigen Verfahren unterlegen!

Komplexe Verfahren:

- Quicksort
- Mergesort

Sortieren durch Zerlegen, Partitionieren (Quicksort): Prinzip:
⇒ Wähle ein beliebiges Element e aus dem Array
⇒ Zerlege das zu sortierende Array in zwei Teile, so dass im linken Teil alle Elemente kleiner e sind und im rechten Teil alle Elemente größer oder gleich e.
⇒ Bearbeite auf diese Weise nun die Teilarrays
⇒ Führe diesen Prozess fort, bis die Teilarrays nur noch ein Element besitzen, damit ist die Sortierung beendet

Anzahl Vergleiche:

- schlechtester Fall: $n+2$ Vergleiche bei einem Durchlauf und Partitionierung immer an den Rand gelegt $\Rightarrow$ Aufwand $O(n^2)$ $\Rightarrow$ Quicksort in diesem Fall ineffizient
- durchschnittlicher Fall Aufwand $O(n \ln n)$

Sortieren durch Zerlegen, Partitionieren (Mergesort): Prinzip:
$\Rightarrow$ Zerlege das zu sortierende Array in zwei Teile, sortiere diese durch weitere Zerlegung
$\Rightarrow$ Füge die sortieren Teile elementweise zusammen, so dass die neue Sequenz sortiert ist
$\Rightarrow$ Führe diesen Prozess fort, bis die Teilarrays nur noch ein Element besitzen, damit ist die Sortierung beendet

Anzahl Vergleiche: $O(n \log n)$

Anmerkung

Sowohl Quicksort als auch Mergesort zerlegen (rekursiv) eine Array in sortierte Teilarrays, die anschließend wieder zusammengefügt werden. Bei Quicksort erfordert das Zerlegen Aufwand, während das Zusammenfügen einfach ist. Im Gegensatz dazu ist bei Mergesort das Zerlegen einfach, aber der Zusammenbau ist aufwändig.

6 Hashverfahren

Ausgangsituation: Elemente eines Datentyps sind nach einem Schlüssel in einem Array gespeichert[1].

Aufgabe: Berechnen des Arrayindex i zu einem gegebenen Schlüssel s als Funktion von s[2]

$\Rightarrow$ Lösung durch Anwendung einer Hashfunktion $h(s)$

Beispiele:

- Verwaltung von Personendaten über das Geburtsdatum
- Verwaltung von Kfz-Daten über das Kennzeichen
- Symboltabellen in Compilern

Definition Hash-Funktion: Sei M eine Menge deren Elemente durch Schlüsselwerte S charakterisiert sind (d.h. jedes Element e aus M besitzt einen Schlüssel s). B sei eine endliche Menge von Behältern, in denen Elemente von M gespeichert werden sollen, mit $|B| = n, n > 0$

Eine Hash-Funktion ist eine totale, d.h. überall definierte Funktion $M \to \{1, \ldots, n\}$

Der berechnete Wert (Nummer des Behälters für e) eines Elements s aus M der Hashfunktion $h(e)$ wird als Hashwert bezeichnet.

Die Gesamtzahl der Behälter $B_1, \ldots, B_n$ wird als Hashtabelle bezeichnet.
Der Wert $n/|M|$ definiert die Schlüsseldichte.

Der Wert n/m ist der Belegungsfaktor der Hash-Tabelle $B_0, \ldots, B_{m-1}$.

Haben unterschiedliche Elemente aus M den gleichen Hashwert, so wird dies als Kollision bezeichnet. Kollisionen kommen häufig vor, da in der Regel $|M| >> n$.

Die Hash-Funktion h sollte die folgenden Eigenschaften haben:

- Sie sollte surjektiv sein, d.h. alle Behälter sollten belegt werden, d.h. es gibt zu jedem Behälter b ein $x \in M$ mit $h(x) = b$
- Die zu speichernden Schlüssel sollen möglichst gleichmäßig über alle Behälter verteilt werden. Jeder Behälter sollte möglichst mit gleicher Wahrscheinlichkeit belegt werden.
- Sie sollte «einfach» zu berechnen sein

[1] verallgemeinert kann statt eines Arrayindex einen Speicheradresse gesucht werden.
[2] weitere Funktionen zum Aufbau («Füllen») und Löschen des Arrays sind notwendig

Wahrscheinlichkeit für Kollisionen? Sei $h : M \to \{1, \ldots, n\}$ eine ideale Hash-Funktion, $e \in M$.

Dann gilt zunächst: $P(h(e) = i) = 1/n$
und für eine Folge von k Schlüsseln, wobei $k < n$, gilt weiter:
$P(\text{Kollision}(1, \ldots, k) = 1 - P(\text{keine Kollision}(1, \ldots, k))$[3]

$P(\text{keine Kollsision}(1, \ldots, k)) = P(1) * P(2) * \ldots * P(k)$
wobei $P(i)$ die Wahrscheinlichkeit ist, dass der i-te Schlüssel einen freien Platz findet und alle vorherigen auch einen freien Platz gefunden haben (d.h. es sind keine Kollisionen aufgetreten)

Es gilt: $P(1) = 1, P(2) = (n-1)/n, P(i) = (n-i+1)/n$, daraus folgt
$$P(\text{Kollision}(1, \ldots, k) = 1 - \frac{n(n-1) * \ldots * (n-k+1)}{n^k}$$

Beispiel: Für die Anzahl der Tage eines Jahres $n = 365$ ergeben sich die folgenden Wahrscheinlichkeiten einer Kollision:
$k = 22$: $P(\text{Kollision}) \approx 0{,}475$
$k = 23$: $P(\text{Kollision}) \approx 0{,}507$
$k = 50$: $P(\text{Kollision}) \approx 0{,}970$

Dies ist das so genannte «Geburtstagsparadoxon»: Sind mehr als 23 Personen zusammen, so haben mit mehr als 50% Wahrscheinlichkeit mindestens zwei von ihnen am selben Tag Geburtstag.

Für Hashverfahren bedeutet die obige Analyse, dass

- Kollisionen praktisch nicht zu vermeiden sind!
- Mit Kollisionen definiert umgegangen werden muss!

6.1 Behandeln von Kollisionen

Hashverfahren, bei denen ein Behälter (theoretisch) beliebig viele Elemente aufnehmen kann, heißen offene Hashverfahren.
(im Gegensatz zu geschlossenen Verfahren, bei denen jeder Behälter nur eine kleine feste Zahl von Elementen beherbergen kann.)

Hashing mit Verkettung: Als Behälter wird eine verkettete Liste verwendet, in die die Elemente gespeichert werden.

[3] P steht hier für Wahrscheinlichkeit

6.2 offene Verfahren, Aufwand

Sei $S = \{x_1, \ldots, x_n\} \subseteq M$ eine zu speichernde Menge und sei HT eine offene Hash-Tabelle der Länge n mit Hash-Funktion h.
$h(e)$ soll in konstanter Zeit ausgewertet werden für alle $e \in M$.

Im Folgenden soll die Rechenzeit wird dann für Operationen Suchen, Einfügen und Löschen in der Hashtabelle betrachtet werden.

Für $0 \leq i < n$ sei $HT[i]$ die Liste der Schlüssel x_j , für die $h(x_j) = i$ gilt. $|HT[i]|$ sei die Länge der i-ten Liste.

Dann kostet jede Operation im schlechtesten Fall max $O(|HT[h(x)]|)$ viele Schritte (über alle möglichen x). $O(|HT[h(x)]|) \leq n$, da maximal n Elemente in einer Liste gespeichert werden.

Damit gilt, dass die Ausführung der Operationen Suchen, Einfügen und Löschen in einer Hash-Tabelle, in der eine Menge S mit n Elementen abgespeichert werden soll, im schlechtesten Fall einen Aufwand $O(n)$ erfordert.

Anmerkung: In der Realität ist der Aufwand geringer, da die Wahrscheinlichkeit, dass alle Elemente in einer Liste «landen», gering ist.

6.2.1 Geschlossenen Hashverfahren

Bei geschlossenen Hashverfahren kann jeder Behälter nur eine konstante Anzahl $a \geq 1$ von Schlüsseln aufnehmen.

Daher ist die Behandlung von Kollisionen in der Regel komplexer (und daher wichtiger) als bei offenen Verfahren.

Im folgenden wird der Fall $a = 1$ betrachtet und Hash-Tabellen, die Schlüssel/Wert-Paare mit Schlüsseln vom Typ String und Werten vom Typ Object speichern. Dabei soll jedem Schlüssel höchstens ein Wert zugeordnet sein.

Kennzeichnung der Felder der Hashtabelle mit einem booleschen Wert. Unterscheidung:

- Behälter wurde noch nie getroffen (true)
- Ein Behälter wurde schon benutzt, ist aber wegen einer vorhergehenden Löschoperation leer. (false)

Grundlegende Idee der Kollisionsbehandlung: Rehashing:

- Neben der «Haupt»-Hashfunktion h_0 werden weitere Hashfunktionen $h_1, \ldots, h_l$ benutzt.
- Für einen Schlüssel x werden dann nacheinander die Behälter $h_0(x), h_1(x), \ldots, h_l(x)$ angeschaut. Sobald ein freier Behälter gefunden wird, kann das Element gespeichert werden.

Problem: Das Auftreten einer freien Zelle für $h_i(x)$ besagt nicht, dass x nicht schon in der Hash-Tabelle enthalten gewesen ist. ⇒ Markieren der gelöschten Paare.

Analog zu den offenen Hashverfahren soll die Folge der Hashfunktionen $h_0, \ldots, h_{n-1}$ so festgelegt werden, dass für jeden Schlüsselwert s sämtliche Behälter $HT[i]$ $(0 \leq i < n)$ erreicht werden. D.h. es gibt eine «gleichmäßige» Verteilung über alle Behälter.

Wahl der Hashfunktionen $h_i(x)$: $h_i(x) := (h(x) + i) \mod n$
Diese einfachste Art der Festlegung wird als lineares Sondieren (linear probing) bezeichnet.

Lineares Sondieren

Eigenschaften:

- Verschieben «kollidierender» Elemente auf den nächsten freien Behälter.
- Bei k hintereinander belegten Behältern gilt: Die Wahrscheinlichkeit, dass der erste freie Behälter nach diesen k Behältern belegt wird, ist mit $(k+1)/m$ wesentlich größer als die Wahrscheinlichkeit, dass ein Behälter im nächsten Schritt belegt wird, dessen Vorgänger noch frei ist. Dadurch entstehen beim linearen Sondieren «Ketten» belegter Behälter.
- Sei $\alpha := n/m$ der Belegungsfaktor einer Hash-Tabelle mit m Behältern, von denen n belegt sind. Beim Hashing mit linearem Sondieren entstehen für eine Suchoperation durchschnittlich folgende Kosten: (Knuth 1973)
  - $(1 + 1/(1-\alpha))/2$ beim erfolgreichen Suchen
  - $1 + 1/(1-\alpha)2)/2$ beim erfolglosen Suchen

Folgerung: Bei Belegung einer Hashtabelle wird Suchen mittels linearem Sondieren ineffizient.
⇒ Alternative Hashverfahren betrachten!

Verallgemeinertes lineares Sondieren: $h_i(x) = (h(x) + c \cdot i) \mod m$
c ist dabei eine ganzzahlige Konstante (> 0), die zu m teilerfremd sein muss, um alle Behälter zu erreichen.

Quadratisches Sondieren: $h_i(x) = (h(x) + i^2) \mod m$
oder für $1 \leq i \leq (m-1)/2)$
$h_{2i-1}(x) = (h(x) + i^2) \mod m$,
$h_{2i}(x) = (h(x) - i^2) \mod m$.
(Wählt man bei dieser Variante $m = 4j+3$ als Primzahl, so wird jeder Behälter getroffen.)

Doppel-Hashing: Seien $h, h* : M \to \{0, \ldots, n-1\}$ zwei Hash-Funktionen.
Dabei seien h und $h*$ so definiert, dass für beide eine Kollision nur mit Wahrscheinlichkeit $1/n$ auftritt, d.h. $P(h(x) = h(y)) = P(h*(x) = h*(y)) = 1/n$
Die Funktionen h und $h*$ heißen unabhängig, wenn eine Doppelkollision nur mit Wahrscheinlichkeit $1/n^2$ auftritt, d.h. $P(h(x) = h(y)$ und $h*(x) = h*(y)) = 1/n^2$.

Eine Folge von Hash-Funktionen wird wie folgt definiert:
Sei $i \geq 1$, dann ist $h_i(x) = (h(x) + h*(x) \cdot i^2) \mod m$.

Problem: Paare von Funktionen finden, die unabhängig sind.

6.3 Hashfunktionen

Im folgenden sollen verschiedene Hashfunktionen vorgestellt werden.

Sei M eine Menge und nat : $M \to \mathbb{N}$ eine Funktion von M in die Menge der natürlichen Zahlen. Dann ist $h(m) = \text{nat}(m) \mod n$ eine Hashfunktion (für n Behälter).

6.3.1 Bildung von Hashwerten aus Wörtern

Sei W die Menge der Wörter aus dem Alphabet $A = \{a, b, \ldots, z\}$. Dann kann ein Wort $w = a_1 a_2 \ldots a_n$ ($a_i \in A$) als eine Zahl $\text{nat}(w) = \text{wert}(a_1)*26^{n-1} + \text{wert}(a_2)*26^{n-2} + \ldots + \text{wert}(a_{n-1})*26 + \text{wert}(a_n)$ aufgefasst werden, wenn man $\text{wert}(a) = 0, \text{wert}(b) = 1, \ldots, \text{wert}(z) = 25$ setzt.

6.3.2 Einfache Bildung von Hashwerten aus Zahlen

Sei $M \subseteq \mathbb{N}$, n eine Menge von Behältern, dann sei $h(x) = x \mod n$

- Alle Behälter werden erfasst
- Nacheinander folgende Schlüssel landen in aufeinanderfolgende Behälter ⇒ Probleme beim Sondieren

Die Mittel-Quadrat-Methode:
Sei $U \subseteq \mathbb{N}$ und sei $k = \sum_{i=0}^{l} z_i * 10^i$
k wird durch die Ziffernfolge $z_l z_{l-1} \ldots z_0$ beschrieben.
Den Wert $h(k)$ erhält man dadurch «Herausgreifen» eines hinreichend großen Blocks aus der Mitte der Ziffernfolge von k^2.

Die mittleren Ziffern von k^2 hängen von allen Ziffern von k ab ⇒ gute Streuung von aufeinander folgenden Werten von k.

Beispiel:
Sei $n = 100$ (Anzahl der Behälter)

k	$k \mod 100$	k^2	$h(k)$
130	30	16900	90
131	31	17161	16
132	32	17424	42