Datenstrukturen und Algorithmen SS 2001

Zusammenfassung und Lernhilfe

David Rybach, Daniel Schneider, Sascha Brandt

Juli 2001

Inhaltsverzeichnis

1 Grundlagen

1.1 Algorithmen und Komplexität

Laufzeit $T(N)$ abhängig (u.a) von Eingabedaten

Prozedur / Rekursion: Beachte Speicherbedarf für Aktivierungsblöcke (enthalten Parameter, Rücksprungadresse, lokale Variablen)

Komplexitätsklassen:

Klasse	Bezeichnung
1	konstant
$\log(\log n)$	doppelt logarithmisch
$logn$	logarithmisch
n	linear
$n \log n$	überlinear
n^2	quadratisch
n^3	kubisch
n^k	polynomiell vom Grad k
2^n	exponentiell
$n!$	Fakultät
n^n	

meistens: maximal polynomiell praktikabel

O-Notation: asymptotisches Verhalten der Laufzeit in engen Schranken unter Vernachlässigung konstanter Faktoren.

- $O(f) \qquad \Omega(f) \qquad \Theta(f) \qquad o(f) \qquad \omega(f)$
- Rechenregeln:
  - *Linearität*:
  $$g(n) = \alpha f(n) + \beta \quad \Rightarrow \quad g \in O(f)$$
  - *Addition*:
  $$f + g \ \in O(\max\{f, g\})$$
  - *Multiplikation*:
  $$a \in O(f) \ \wedge \ b \in O(g) \quad \Rightarrow \quad a \cdot b \in O(f \cdot g)$$
  - *Grenzwert*:
  $$\text{Ex.} \lim_{n\to\infty} \frac{g(n)}{f(n)} \quad \Rightarrow \quad g \in O(f)$$
  - Θ:
  $$\Theta(f) = \Omega(f) \cap O(f)$$

1.2 Datenstrukturen

Elementare Datentypen	*zusammengesetzte Datentypen*	*Pointer*
(int, cardinal) Operationen durch Hardware-Instruktionen	(array, record) Konstanten Aufwand für Zugriff	Referenz Zugriffsaufwand abhängig von der Größe der Struktur
Benutzerdefinierte Datentypen (list, stack, queque, tree)		

1.2.1 Listen

- *Operationen*:
  `insert`, `delete`, `read`
- *Implementierung mit Zeigern*:
  Darstellung Anfang / Ende: Zeiger `head` → 1. El.; letztes El. → `z`
  `proc init = head^.next = z; z^.next = z;`
- *Implementierung mit Arrays*:
  - sequentielles Array
  - Cursor-Darstellung
  - vgl. Pointer-Implementierung, zweites Array mit Indizes für Nachfolgeelemente

1.2.2 Stacks (LIFO)

- Speicherung in seq. Ordung (vgl. Listen), aber nur Zugriff auf 1. Element
- *Operationen*:
  `Push`: Neues Element am oberen Ende (Top) anfügen
  `Pop`: Oberstes Element entfernen und zurückgeben
- *Implementierung mit Zeigern*: gleiche Datenstrukturen wie Liste, aber andere Operationen
- *Implementierung mit Arrays*:

	← `top`
a_2	
a_1	
a_0	

`top` bei jedem Zugriff verändern, Abfangen von Über- / Unterlauf

1.2.3 Queues (FIFO)

- *Operationen*:
  `Put`: Neues Element am Ende der Queue anfügen
  `Get`: Vorderstes Element entfernen und zurückliefern
- Abfangen von Über- / Unterlauf notwendig

1.2.4 Bäume

- Menge von Knoten mit Relation, die Hierarchie definiert
- *Grad* eines Knoten: Zahl seiner unmittelbaren Nachfolger
- *Pfad*: Folge von unmittelbar aufeinander folgenden Knoten, Länge = Zahl der Knoten -1 = Zahl der Kanten
- *Tiefe / Höhe* eines Knoten: Pfadlänge bis zur Wurzel
- *Tiefe / Höhe* eines Baums: Pfadlänge Wurzel - tiefstes Blatt
- *Grad* des Baums: Maximum der Grade seiner Knoten
- *Darstellung*: geschachtelte Menge, Einrückung, geschachtelte Klammerung
- *min / max Baumhöhe*:
  T Baum, Grad $d \geq 2$, n Knoten:
  max Höhe = $n-1$
  min Höhe = $\lceil \log d\ (n(d-1)+1) \rceil - 1 \ \leq\ \lceil \log_d n \rceil$
  Binärbaum: $h = \lceil \mathrm{ld}\ (n+1) \rceil - 1 \ \leq\ \lceil \mathrm{ld}\ n \rceil$
- *Implementierung*:
  - Pointer-Darstellung:

    ```
    TYPE node = REF RECORD
                      key : ItemType;
                      left, right : node;
                    END;
    ```

    Wurzel, Blätter: `root`, `z` (vgl. Listen)
  - Array-Einbettung:
    * Gut für Darstellung vollständiger Bäume (alle Ebenen bis auf die letzte sind vollständig gefüllt, letzte Ebene von links nach rechts gefüllt)
    * Durchnummerierung der Knoten: ebenenweise, von links nach rechts
    * Vorgänger: `pred` = $\lfloor n/2 \rfloor$
    * Nachfolger:

      $$\texttt{succ} = \begin{cases} 2n & \text{linker Sohn} \\ 2n+1 & \text{rechter Sohn} \end{cases}$$
  - Array von Poinern ($d > 2$)
    * ```
      TYPE node = REF RECORD
                        key : ItemType;
                        sons : ARRAY [1..d] OF node;
                      END;
      ```
    * Nachteil: maximaler Grad vorgegeben, Platzverschwendung bei starker Gradvariation
  - Pseudo-Binärbaum
      ```

    * Jeder Knoten verweist nur auf sein erstes Kind und seinen rechten Bruder
    * Array-Einbettung
      · ebenenweises Durchnummerieren der Knoten, Ablegen in einem Array → Geschwister-Knoten in aufeinanderfolgenden Positionen
      · Zusätzliche Arrays zum Auffinden von Paaren / Left - / Rightmostchild
      · Durchlauf aller Nachfolger von i:
        `leftmc`(i) ... `rightmc`(i)

1.2.5 Travesierung eines Baumes (Durchlaufen)

- Kompletter Durchlauf
- Tiefendurchlauf:
  zu Knoten n werden rekursiv seine Teilbäume $n_1, \ldots, n_k$ durchlaufen
  - *Preorder*: betrachte n, durchlaufe $n_1, \ldots, n_k$
  - *Postorder*: durchlaufe $n_1, \ldots, n_k$, betrachte n
  - *Inorder*: durchlaufe n_1, betrachte n, durchlaufe $n_2, \ldots, n_k$
- Breitendurchlauf (*Levelorder*):
  Listen der Knoten nach ihrer Tiefe, bei gleicher Tiefe von links nach rechts
- Implementierung:
  - rekursiv
  - iterativ
    * mit Stack (preoder)
    * mit Queue (levelorder)

1.3 Entwurfsmethoden

- *Divider & Conquer*: top-down
- *Dynamische Programmierung*: bottom-up
- *Memoization*: Speicherung von Zwischenwerten zur Vermeidung von Ineffizienz bei Rekursionen

1.4 Rekursionsgleichungen

- sukzessives Einsetzen
- Master-Theorem

$$\text{Form}: \quad T(n) = \begin{cases} 1 & n = 1 \\ a \cdot T\left(\frac{a}{b}\right) + d(n) & n > 1 \end{cases}$$

mit $a \geq 1$, $b > 1$, $n \mapsto d(n)$
$\Rightarrow$ für $d(n) \in (n^\gamma)$ mit $\gamma > 0$:

$$T(n) \in \begin{cases} O(n^\gamma) & a < b^\gamma \\ O(n^\gamma \log_b n) & a = b^\gamma \\ O(n^{\log_b a}) & a > b^\gamma \end{cases}$$

2 Sortieren

2.1 Definitionen

2.1.1 Partielle Ordung

Es sei $\mathcal{M}$ eine nicht leere Menge und $\leq \subseteq \mathcal{M} \times \mathcal{M}$ eine binäre Relation auf $\mathcal{M}$. Das Paar $(\mathcal{M}, \leq)$ heißt eine *partielle Ordnung auf* $\mathcal{M}$ genau dann wenn $\leq$ die folgenden Eigenschaften erfüllt:

- **Relflexivität**: $x \leq x \qquad \forall x \in \mathcal{M}$
- **Transitivität**: $x \leq y \ \wedge \ y \leq z \Rightarrow x \leq z \qquad \forall x, y, z \in \mathcal{M}$
- **Antisymmetrie**: $x \leq y \ \wedge \ y \leq x \Rightarrow x = y \qquad \forall x, y \in \mathcal{M}$

2.1.2 Strikter Anteil einer Ordungsrelation

Für eine partielle Ordung $\leq$ auf einer Menge $\mathcal{M}$ definieren wir die Relation $<$ durch:

$$x < y \ := \ x \leq y \ \wedge \ x \neq y$$

Die Relation $<$ heißt auch der *strikte Anteil von* $\leq$.

2.1.3 Totale Ordnung

Es sei $\mathcal{M}$ eine nicht leere Menge und $\leq \subseteq \mathcal{M} \times \mathcal{M}$ eine binäre Relation über $\mathcal{M}$. $\leq$ heißt eine *totale Ordung auf* $\mathcal{M}$ genau dann wenn gilt:

- $(\mathcal{M}, \leq)$ ist eine partielle Ordung und
- **Trichotomie**: $x < y \ \dot{\vee} \ x = y \ \dot{\vee} \ y < x \qquad \forall x, y \in \mathcal{M}$

2.1.4 Sortierproblem

Gegeben sei eine Folge `a[1]`,...,`a[N]` von Records mit einer Schlüsselkomponente `a[i].key` (i=1,...,N) und eine totale Ordnung $\leq$ auf der Menge aller Schlüssel. Das *Sortierproblem* besteht darin, eine Permutation π der ursprünglichen Folge zu bestimmen, so dass gilt:

$$\texttt{a}[\pi_1]\texttt{.key} \ \leq \ \texttt{a}[\pi_2]\texttt{.key} \ \leq \ \ldots \leq \ \texttt{a}[\pi_{N-1}]\texttt{.key} \ \leq \ \texttt{a}[\pi_N]\texttt{.key}$$

2.1.5 Unterscheidungskriterien

- Sortiermethode
- Effizienz [1]
- *intern* / *extern* (Records im Arbeitsspeicher / Platten)
- *direkt* / indirekt (Pointer / Array-Indizes)
- im Array oder nicht im Array
- *in situ* (ein Array ohne zusätzliches Hilfsfeld)
- *allgemein* / *speziell* [2]
- *stabil* (Reihenfolge von Records mit gleichen Keys bleibt erhalten)

2.2 Elementare Sortierverfahren

2.2.1 SelectionSort

Sortieren durch Auswahl

Prinzip: Im i-ten Durchlauf der Schleife:

- Bestimme den Datensatz mit dem kleinsten Schlüssel aus `a[i],...,a[N]`
- Vertausche dieses Minimum mit `a[i]`

Algorithmus:

```
FOR i :=1 TO N-1
     min := min_{i<j≤N} a[j]
     Tausche a[min] ↔ a[i]
END;
```

Komplexität: $N-1$ Schleifendurchläufe, pro Schleifendurchgang i eine Vertauschung ($\hat{=}$ 3 Bewegungen und $N-i$ Vergleiche). $\Rightarrow$

- $3 \cdot (N-1)$ Bewegungen
- $\frac{N \cdot (N-1)}{2}$ Vergleiche

Vorteil: Jeder Datensatz wird nur einmal bewegt $\Rightarrow$ besonders geeignet für Sortieraufgaben mit großen Datensätzen.

[1] $O(N^2)$ für elementare, $O(N \log N)$ für höhere Sortierverfahren

[2] speziell: z.B. BucketSort

2.2.2 InsertionSort

Sortieren durch Einfügen

Prinzip: Im i-ten Durchgang der Schleife ($i = 2, \ldots, N$). Teilfolge `a[1],...,a[i-1]` bereits sortiert.

- Füge den Datensatz `a[i]` an der korrekten Position der bereits sortierten Teilfolge ein.

Algorithmus:

```
FOR i := 2 TO N
    v := a[i]; j := i;
    WHILE (a[j-1] > v) AND (j > 1)
        a[j] := a[j-1];
        DEC j;
    END;
    a[j] := v;
END;
```

Komplexität:

- *Best Case* (Vollständig vorsortiert): $N-1$ Vergleiche; $2(N-1)$ Bewegungen
- *Worst Case* (Umgekehrt sortiert):

$$\frac{N^2}{2} + \frac{N}{N} - 1 \text{ Vergleiche} \qquad \frac{N^2}{2} \text{ Bewegungen}$$

- *Average Case*: $\approx \frac{N^2}{4}$ Vergleiche, $\approx \frac{N^2}{4}$ Bewegungen.

Vorteil: Für "fast sortierte" Folgen fast lineares Verhalten. Kann vorhandene Ordung besser ausnutzen

2.2.3 BubbleSort

Sortieren durch wiederholtes Vertauschen von benachbarten Array-Elementen

Prinzip: Im i-ten Durchlauf der Schleife ($i = N, N-1, \ldots, 2$):

- Schleife j=2,3,...,i: ordne `a[j-1]` und `a[j]`

Algorithmus:

```
FOR i := N TO 1
    FOR j := 2 TO i
            IF a[j-1] > a[j]
            Tausche a[j] ↔ a[j-1]
    END;
END;
```

Komplexität: Anzahl der Vergleiche ist unabhängig von der Vorsortierung

- $\frac{N(N-1)}{2}$ Vergleiche
- *Best Case*: 0 Bewegungen
- *Worst Case*: $\approx 3N^2/2$ Bewegungen
- *Average Case*: $\approx 3N^2/4$ Bewegungen

Vorteil: *Kein* Vorteil, da immer $N^2/2$ Vergleiche $\Rightarrow$ ineffizient.

2.2.4 Fazit

Einsatzgebiete: InsertionSort für fast sortierte Folgen, SelectionSort für große Datensätze, BubbleSort nie. InsertionSort und SelectionSort sollten nur für Sortierprobleme mit $N \leq 50$ eingesetzt werden, sonst höherere Verfahren.

2.2.5 Indirektes Sortieren

Bei Sortierproblemen mit sehr großen Datensätzen großer Rechenaufwand für das Vertauschen von Records. Deshalb sortiert man nur Verweise auf die Datensätze.

Verfahren:

- Index-Array `p[1..N]` mit `p[i] = i`
- Für Vergleiche erfolgt Zugriff auf einen Record: `a[p[i]]`
- Vertauschen der Indizes statt der Array-Elemente
- Evtl. werden nach dem Sortieren die Records selbst umsortiert ($O(N)$)
  - Permutation mit zusätzlichen Array b: `b[i] := a[p[i]]`
  - *In situ, in place*: Ohne Zusätzliches Array:
    - Wenn `p[i] = i` $\surd$
    - Wenn `p[i] # i` zyklisch vertauschen:
      1. Record kopieren `t := a[i]` $\Rightarrow$ "Loch" an Position i
      2. Iterieren

2.2.6 BucketSort

Voraussetzung: Schlüssel können als ganzzahlige Werte im Bereich 0,...,M-1 dargestellt werden, so dass sie als Array-Index verwendet werden können.

Prinzip:

- Erstelle Histogramm (zähle Häufigkeit von Keys)
- Berechne aus Histogramm die Postion für jeden Record
- Bewege die Records an die richtige Postition

Algorithmus:

```
Initialisiere count[]
Erstelle Histogramm
FOR i:= 1 TO M-1
    count[j] := count[j-1] + count[j];
FOR i := N TO 1
    b[count[a[i]]] := a[i];
    DEC count[a[i]];
b[] in a[] kopieren
```

Komplexität: $T(N) = O(\max\{N, M\})$

Eigenschaften: Stabil, Arbeitet nicht in situ

2.3 Höhere Sortierverfahren

2.3.1 QuickSort

Prinzip: Folgt dem Devide-and-Conquer-Ansatz

- Partitioniere das Array `a[l..r]` bzgl. eines Pivot-Element `a[k]` in zwei Teilarrays `a[l..k-1]` und `a[k+1..r]`, so dass gilt

$$\mathtt{a[i]} \leq \mathtt{a[k]} \qquad \forall i \in \{l, \ldots, k-1\}$$

$$\mathtt{a[k]} \leq \mathtt{a[j]} \qquad \forall i \in \{k+1, \ldots, r\}$$

- *Rekursion*: linkes `a[l ...k-1]` und rechtes Teilarray `a[k-1 ...r]` bearbeiten

Algorithmus:

```
Quicksort (l, r : INT)
    IF l < r THEN
        k := Partition (l, r);
        Quicksort (l, k-1);
        Quicksort (k+1, r);
    END;
END;


Partition (l, r : INT)
    PivotElement := a[r];
    REPEAT
        suche i von links mit a[i] >= PivotElement;
        suche j von rechts mit a[j] <= PivotElement;
        tausche a[i] <-> a[j];
    UNTIL Zeiger kreuzen
```

```
    a[i] und a[j] rückvertauschen
    a[i] und a[r] vertauschen

    RETURN i als Position des PivotElements;
END;
```

Komplexität: Für obige Implentierung:

- *Best Case*: $T(N) = (N+1) \cdot \mathrm{ld}(N+1)$
- *Average Case*: $T(N) = 1.386 \cdot (N+1) \cdot \mathrm{ld}(N+1)$
- *Worst Case*:
$$T(N) = \frac{(N+1)\cdot(N+2)}{2} - 3$$

Eigenschaften:

- Effizienz durch nur einen Schlüsselvergleich in der innersten Schleife
- Durch Rekursionsoverhead nicht geignet für kleine Folgen
- Verbesserungen [3]
  - PivotElement := `(a[l] + a[r] / 2)`
  - PivotElement := kleinstes Element von drei zufällig gewählten Elementen
  - Ineffizienz bei kleinen Arrays kann behoben werden durch anwenden von z.B. InsertionSort ab einer TeilArray-Größe von z.B. 12 oder 22
  - Minimierung des Speicherplatzbedarfs für die Rekursion: Das kleinere Teilarray zuerst bearbeiten
  - Iterativ implentieren (erfordert Stack)

2.3.2 HeapSort

Definition Heap, Heap-Eigenschaft: Ein Heap ist ein links-vollständiger Binärbaum, der in einem Array eingebettet ist:

- Ein Array `a[1 ...N]` erfüllt die *Heap-Eigenschaft*, wenn gilt:
$$\mathtt{a}\left[\left\lfloor\frac{i}{2}\right\rfloor\right] \geq \mathtt{a[i]} \qquad \forall i = 2,\ldots,N$$
- Ein Array `a[1 ...N]` ist ein *Heap* beginnend an Position $l = 1,\ldots,N$, falls:
$$\mathtt{a}\left[\left\lfloor\frac{i}{2}\right\rfloor\right] \geq \mathtt{a[i]} \qquad \forall i = \underline{2l},\ldots,N$$

Jedes Array `a[1 ...N]` ist ein Heap beginnend in Positon $l = \lfloor N/2 \rfloor + 1$

[3] siehe auch Implementierung in Sorter v0.6, (*www.rybach.de)*

Algorithmus:

1. Wandle das Array `a[1 ...N]` in einen Heap um.
2. `FOR i := 1 TO N-1`
   a) Tausche `a[1]` (Wurzel) ↔ `a[N-i-1]`
   b) Stelle für das Rest-Array `a[1 ...(N-i)]` die Heap-Eigenschaft wieder her
3. Algorithmus für Heap-Eigenschaft:

```
DownHeap (i, k : INT)      v := a[i]
     LOOP
        Wenn a[i] Blatt → RETURN
        Bestimme, wenn existiert, größeren der beiden Söhne → j
        Wenn beide kleiner als v → RETURN
        a[i] := a[j]; i:= j;
    END;
     a[i] := v;
END;
```

Komplexität: HeapSort sortiert die Folge `a[1 ...N]` mit höchstens

$$2N\text{ld}(N+1) - 2N$$

Vergleichen.

2.3.3 MergeSort

MergeSort sortiert nach der *Devide-and-Conquer-Strategie*.

Komplexität:

- *worst case* = *average case*: $N\text{ld}N$

Eigenschaften: zusätzlicher Speicherplatz $O(N)$, nicht in situ, aber sequentiell

2.3.4 Untere und obere Schranke für das Sortierproblem

Obere Schranke: $T_{\mathcal{A}}(N)$:= Zahl der Schlüsselvergleiche um eine N-elentige Folge von Schlüsselelementen mit dem Algorithmus $\mathcal{A}$ zu sortieren.

$$T_{\mathcal{A}}(N) \leq N\lceil \text{ld}N \rceil - N + 1$$

Untere Schranke: $T_{\min}(N)$:= Zahl der Vergleiche für den effizientesten Algorithmus

$$T_{\min}(N) \geq N\ \text{ld}N - N\ \text{ld}\ e$$

3 Suchen in Mengen

3.1 Problemstellung

- *Gegeben*: Menge von Records (key + weitere Komponenten), ohne Duplikate (bzgl. Schlüssel oder voller Record)
- *Typische Aufgabe*: Finde zu einem gegebenen Key den Record und führe ggf. eine Operation aus
- Hier primär Betrachtung des *Dictionary-Problem*
- *Notationen*:
  Universum: U := Menge aller möglichen Schlüssel
  Menge: $S \subseteq U$
- **Wörterbuch-Operationen**:
  - `Search (x, S)`: Falls $x \in S$, liefere den vollen zu x gehörigen Record, sonst Meldung "$x \notin S$".
  - `Insert (x, S)`: Füge Element x zur Menge X hinzu: $S := S \cup \{x\}$; Fehler, wenn $x \in S$
  - `Delete (x, S)`: Entferne Element x aus der Menge S: $S := S \setminus \{x\}$; Fehler, wenn $x \notin S$

3.2 Einfache Implementierung

3.2.1 Ungeordnete Arrays und Listen

Darstellung $\rightarrow$ Vergleiche 1.2.1

- Liste im Array
- verkettete Liste

3.2.2 Vergleichsbasierte Methoden

Voraussetzung: Existenz einer Ordungsrelation
Betrachtung von Daten im geordneten Array `S` mit `S[i] < S[i+1]` für $1 \leq i \leq n-1$.

Allgemeines Programmschema

```
S : ARRAY [0 ...n+1] OF element;
S[0] := −∞;
S[n+1] := +∞;

PROCEDURE Search (a, S)
VAR low, high : element;
BEGIN
    low := 1; high := n;
    next := zahl ∈ [low..high]
    WHILE (a # S[next]) AND (low < high) DO
        IF a < S[next] THEN
            high := next -1;
        ELSE low := next +1;
        next := zahl ∈ [low..high]
    END;
        IF a = S[next] THEN
            (* a an Pos next gefunden *)
        ELSE
            (* a  nicht gefunden *)
```

3.2.3 Bitvektordarstellung (Kleines Universum)

Annahme : $N = |U|$ = vorgegebene maximale Anzahl von Elementen
$S \subset U = \{0, 1, \ldots, N-1\}$

Methode : Key = Index im Array: `ARRAY OF BOOLEAN`

- `Bit[i] = FALSE`, $i \notin S$
- `Bit[i] = TRUE`, $i \in S$

Komplexität :

- Operationen $O(1)$
- Init $O(N)$
- Platz $O(N)$

3.2.4 Spezielle Array-Implemtierung

Prinzip : Bitverktor-Darstellung mit zwei Hilfsarrays ohne $O(N)$-Initialisierung

Deklarationen :

- `Bit[0 ...N-1] : ARRAY OF BOOLEAN` Bitvektor
- `Ptr[0 ...N-1] : ARRAY OF INTEGER` Pointer-Array

- `Stk[0 ...N-1] : ARRAY OF INTEGER` Stack-Array

Methode : Invariante: $i \in S \quad \Leftrightarrow$

- `Bit[i] = TRUE` $\wedge$
- $0 \leq$ `Ptr[i]` $\leq top \quad \wedge$
- `Stk[Ptr[i]] = i`

Anfangs: $top = -1$

Komplexität :

- Platz: $O(N)$
- Alle Standard-Operationen: $O(1)$

3.3 Hashing

3.3.1 Begriffe und Unterscheidung

- offenes / geschlossenes Hashing
- Kollisionsstrategien:
  - lineares Sondieren
  - quadratisches Sondieren
  - Doppelhashing
- Hash-Funktionen
- erweiterbares Hashing in Verbindung mit Hintergrundspeicher

3.3.2 Prinzip

- Es stehen m Speicherplätze in *Hashtabelle* T zur Verfügung:
  `T : ARRAY [0..m-1] OF element`
- Key $x \in U = \{0, \ldots, N-1\}$ wird mittles *Hashfunktion* $h(x)$ auf Speicherplatz in T abgebildet:

$$\begin{array}{rcl} h\,:\,U & \rightarrow & \{0,1,\ldots,m-1\} \\ x & \mapsto & h(x) \end{array}$$

  x wird in `T[` $h(x)$ `]` gespeichert, falls dieser Platz frei ist.
- In der Regel $m \ll N \Rightarrow$ *Kollisionen*:

$$h(x) \;=\; h(y) \quad \text{fuer } x \neq y$$

- Bei Kollision: entweder Verweis auf andere Adresse (offenes Hashing 3.3.4), oder Ermittlung eines anderen Speicherplatzes mittles *Sondierungsfunktion* (geschlossenes Hashing 3.3.5)
- Bei `search (x, S)`: zunächst $h(x)$ berechnen, dann x in `T[` $h(x)$ `]` suchen

3.3.3 Hashfunktionen

Anforderungen

- $h(x)$ surjektiv $\Rightarrow$ ganze Hashtabelle wird abgedeckt
- Schlüssel sollen gleichmässig verteilt werden
- Berechnung soll effizient sein (kein hoher Rechenaufwand)

Divisions-Rest-Methode

Sei m die Größe der Hashtabelle

$$h \; : \; x \mapsto x \bmod m$$

Wähle m möglichst so:

- m prim
- m teilt nicht $2^i \pm j$, wobei i, j kleine Zahlen $\in \mathbb{N}_0$

Nachteil: Aufeinanderfolgende Schlüssel werden auf aufeinanderfolgende Speicherplätze abgebildet $\Rightarrow$ *Clustering*

Mittel-Quadrat-Methode

Ziel: Auch nahe beieinanderliegende Schlüssel auf die ganze Hashtabelle verteilen.

$$h(x) = \text{ mittlerer Block von Ziffern von } x^2$$

Beispiel (m = 100):

x	$x \bmod 100$	x^2	$h(x)$
127	27	16**12**9	12

3.3.4 Offenes Hashing

- Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt.
- Ein Array von Zeigern verwaltet die Behälter:
  `HashTable : ARRAY [0..m-1] OF REF ListElement`
- *Belegungsfaktor*: $\alpha = \frac{n}{m}$
- Komplexität
  - Adresse berechnen, Behälter aufsuchen: $O(1)$
  - Liste durchsuchen:
    * Average Case (erfolgr. Suche): $O(1 + \frac{\alpha}{2})$
    * Worst Case (erfolglose Suche): $O(\alpha) = O(n)$
  - Platz: $O(m + n)$

3.3.5 Geschlossenes Hashing

- Arbeitet mit statischem Array:
  `HashTable : ARRAY [0..m-1] OF Element`
- Kollisionen werden mit erneuter Adressberechnung behandelt, mittels *Sondierungsfunktion*
- Sondierungsfunktionen $h(i,x)$:

  Lineares Sondieren :

  $$h(i,x) = (h(x) + c \cdot i) \bmod m \qquad 1 \leq j \leq m-1;\ c \in \mathbb{N}$$

  c und m sollten teilerfremd sein.
  Linares Sondieren tendiert zum *primary clustering* (Kettenbildung) $\Rightarrow$ ineffizient

  Quadratisches Sondieren :

  $$h(i,x) = (h(x) + i^2) \bmod m \qquad 0 \leq i \leq m-1$$

  Primäres Clustering wird verhindert, es enstehen also keine langen Ketten, Sekundäres Clustering tritt trotzdem auf, d.h. Schlüssel mit gleichem Hashwert werden auf die gleiche Sondierungsbahn gebracht.

  Doppelhashing :

  $$h(x,i) = (h(x) + h'(x) \cdot i^2) \bmod m$$

  Wobei $h(x)$ und $h'(x)$ unabhängig sind.
  Eigenschaften:

  - Die mit Abstand beste Kollisionsstrategie
  - Kaum unterscheidbar von idealem Hashing

3.3.6 Zusammenfassung Hashverfahren

- Average Case: allgemein effizientes Verhalten ($O(1)$)
- Worst Case: Operationen mit $O(n)$
- Nachteil: Alle Operationen, die auf einer Ordung basieren werden nicht unterstützt.
- Anwendungen, wenn $|U| \gg |S|$ und dennoch ein effizienter Zugriff auf die Elemente wünschenswert ist.

3.4 Binäre Suchbäume

3.4.1 Allgemeine binäre Suchbäume

Definition

Ein binärer Suchbaum für die n-elementige Menge $S = \{x_1 \leq x_2 \leq \ldots \leq x_n\}$ ist ein binärer Baum mit n Knoten $\{v_1 \ldots v_n\}$. Die Knoten sind mit den Elementen von S beschriftet, d.h. es gibt ein injektive

Abbildung *Inhalt* : $\{v_1 \dots v_n\} \rightarrow S$. Die Beschriftung bewahrt die Ordnung, d.h. wenn v_i im linken Unterbaum, v_j im rechten und v_k Wurzel des Baums ist, dann gilt:

$$Inhalt[v_i] < Inhalt[v_k] < Inhalt[v_j]$$

Operationen

Search :

```
Search (a, S)
   v := Root of T;
   WHILE (v is node) AND (a ≠ Inhalt[v]) DO
      IF a < Inhalt[v] THEN
         v := LeftSon[v];
      ELSE
         v := RightSon[v];
```

Insert : Gehe entsprechend der Binärbaum-Ordung durch den Baum, bis zu einem Blatt, dahinter wird eingefügt

Delete : Fallunterscheidung, wenn a der zu entfernende Key ist und v der Knoten mit $Inhalt[v] = a$:

1. v ist Blatt
   - streiche v aus dem Baum
2. v hat mindestens ein Blatt als Sohn
   - ersetze v durch den anderen Sohn
   - streiche v und das Blatt aus dem Baum
3. v hat zwei Söhne die innere Knoten sind
   - ersetze v mit dem rechtesten Unterknoten im linken Unterbaum von $v \rightarrow w$
   - entferne w, mit Verfahren 2

Zeitkomplexität

- *Search, Insert, Delete*: $O(h(T))$ mit $h(T)$ = Höhe des Baums T
- *ListOrder*: = $O(|S|) = O(n)$

3.4.2 Der optimale binäre Suchbaum

Diese Bäume sind gewichtet mit der Zugriffsverteilung, die die Häufigkeit (oder Wichtigkeit) der Elemente von S wiederspiegelt.

Definition Zugriffsverteilung

Sei $S = \{x_1 \leq x_2 \leq \ldots \leq x_n\}$ und x_0, x_{n+1} Sentinels mit $x_0 \leq x_i \leq x_{n+1}$ $\forall i = 1, \ldots, n$.
Die Zugriffsverteilung ist ein $(2n+1)$-Tupel $(\alpha_0, \beta_1, \alpha_1, \beta_2, \alpha_2, \ldots, \alpha_n, \beta_n)$ von Wahrscheinlichkeiten, für das gilt:

- $\beta_i \geq 0$ ist die Wahrscheinlichkeit, dass eine Search(a, S)-Operation *erfolgreich* im Knoten $x_i = a$ endet

- $\alpha_i \geq 0$ ist die Wahrscheinlichkeit, dass eine Search(a, S)-Operation *erfolos* mit $a \in (x_i, x_{i+1})$ endet

Konstruktion

Mittels *Bellmann-Knuth-Algorithmus* (s. Skript)

3.5 Balancierte Bäume

Balance-Kriterien:

- *Gewichtsbalancierte Bäume* (BB-Bäume): Anzahl der Blätter in den Unterbäumen möglichst gleich

- *Höhenbalancierte Bäume*: Höhenunterschied der Unterbäume möglichst gering. Untertypen:

3.5.1 AVL-Bäume

Definition

Ein AVL-Baum ist ein binärer Suchbaum mit einer Struktur Invarianten: Für jeden Knoten gilt, dass sich die Höhen seiner beiden Teilbäume höchstens um eins unterscheiden.

Höhe eines AVL-Baums

Sei $N(h)$ die minimale Anzahl der Knoten in einem AVL-Baum der Höhe h.

$$N(h) = 1 + N(h-1) + N(h-2)$$

$$N(h) \leq n \leq N(h+1)$$

Operationen

Um die Struktur-Invariante eines AVL-Baums auch nach eine Update-Operation wiederherzustellen, benötigt man *Rebalancierungs-Operationen*. Dabei wird die Balance rückwärts auf dem ganzen Pfad vom veränderten Knoten bis zur Wurzel durchgeführt. Ist die Balance an einer Stelle gestört, kann sie mittels *Rotation* oder *Doppelrotation* wiederhergestellt werden.

Rotation : Von einem betroffenen Knoten betrachtet man nur den von ihm induzierten Teilbaum. Einfache Rotation muß durch geführt werden, wenn der betroffene (höhere) Teilbaum *außen* liegt.
Der betroffene Knoten rotiert zum kürzesten Teilbaum hinunter und übernimmt den innersten Knoten des heraufrotierenden Knotens als inneren Sohn.

Doppelrotation : Eine Doppelrotation muß durchgeführt werden, wenn der betroffene Teilbaum *innen* liegt. Es wird zunächst eine Außen-Rotation im Vaterknoten der Wurzel des betroffenen Teilbaums durchgeführt, und anschließend eine Rotation in entgegengesetzter Richtung im Vaterknoten dieses Knotens.

Komplexität

- Balance-Überprüfung: $O(\text{ld } n)$
- (Doppel-) Rotation: $O(1)$
- Standard-Operationen: $O(\text{ld } n)$
- Platzkomplexität: $O(n)$

3.5.2 (a, b)-Bäume

Prinzip

Bei einem (a, b)-Baum haben alle Blätter gleiche Tiefe. Die Zahl der Söhne eines Knotens liegt zwischen a und b.

Speicherung einer Menge als (a, b)-Baum

$S = \{x_1 < \ldots < x_n\} \subseteq U$ geordnete Menge, T leerer (a, b)-Baum mit n Blättern. Dann wird S in T so gespeichert:

1. Die Elemente von S werden in den Blättern w von T entsprechen ihrer Ordung von links nach rechts gespeichert.
2. Jedem Knoten V werden die $\rho(v)$ (= Anz. Söhne von v) -1 Elemente $k_1(v) < k_2(v) < \ldots < k_{\rho(v)-1}(v) \in U$ so zugeordnet, dass für alle Blätter w im i-ten Unterbaum von v mit $1 < i < \rho(v)$ gilt:
$$k_{i-1}(v) < \mathit{Inhalt}[w] \leq k_i(v)$$

Für einen (a, b)-Baum mit n Blättern und Höhe h läßt sich folgendes ableiten:

$$\begin{array}{lcl} 2a^{h-1} & \leq n \leq & b^h \\ \log_b n & \leq h \leq & 1 + \log_a \frac{n}{2} \end{array}$$

Speicherausnutzung

Jeder Knoten muß $(2b-1)$ Speicherzellen besitzen, mit b Zeigern auf Söhne und $b-1$ Schlüsseln.

Standard-Operationen

- Search: Pfad von der Wurzel zum Blatt auswählen, über Elemente $k_1(v), \ldots, k_{\rho(v)-1}(v)$ an jedem Knoten v
- Insert: Nach `Insert` müssen evtl. wiederholt Knoten gespalten werden, falls diese zu voll sind.
- Delete: Nach `Delete` müssen evtl. Knoten *verschmolzen* oder *gestohlen* werden, um den Baum zu rebalancieren

Alle Operationen haben Komplexität von $O(\log n)$ mit $n = |S|$