1 Grundlagen

Ein Algorithmus ist ein mit formalen Mitteln beschreibbares, mechanisch nachvollzeihbares Verfahren zur Lösung einer Klasse von Problemen

- Dijkstra: Man kann durch Testen zwar die Anwesenheit, nicht aber die Abwesenheit von Fehlern zeigen
- Maß für Effizienz: benötigter Speicherplatz und benötigte Rechenzeit (best case, worst case, average case)
- Divide-and-Conquer-Verfahren (z.B. Polynomprodukt)
  1. Divide: Teile das Problem der Größe N in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn N > 1
  2. Conquer: Löse die Teilprobleme auf dieselbe Art (rekursiv)
  3. Merge: Füge Teillösungen zur Gesamtlösung zusammen
- Komplexität von Algorithmen (O-Notation)

  Aufwand oder Komplexität eines Algorithmus gibt die Anzahl der problemrelevanten Elementaroperationen bei seiner Ausführung an (hängt von der Problemgröße n ab)

  1. O(f) = $\{g : \mathbb{N} \to \mathbb{R} \mid \exists\, n_0 \in \mathbb{N}, \exists\, c > 0 : \forall n \geq n_0 : g(n) \leq c * f(n)\}$
  2. Ω(f) = $\{g : \mathbb{N} \to \mathbb{R} \mid \exists\, n_0 \in \mathbb{N}, \exists\, c > 0 : \forall n \geq n_0 : g(n) \geq c * f(n)\}$
  3. Θ(f) = O(f) $\cap \Omega$(f)

  g ist höchstens | mindestens | genau von der Größenordnung f,

  falls g $\in$ O(f) | g $\in \Omega$(f) | g $\in \Theta$(f)

  Es gilt = log_2n $\subset$ O(n) $\subset$ O($N * log_2 n$) $\subset$ O(n^2) $\subset$ O(2^n)
- $N * logN \in O(N^2)$

  $lim_{n\to\infty} \frac{NlogN}{N^2} = lim_{n\to\infty} \frac{logN}{N}$ (l'Hopital) = $lim_{n\to\infty} \frac{1/N}{1} = 0$

  $lim \frac{g(n)}{f(n)} < c$ g $\in$ O(f)

  In O(f) liegen alle Funktionen, die abgesehen von einer Konstanten für n gg. unendlich kleiner als f sind
- Was ist Komplexität?

  Anzahl der Elementaroperationen

- Polynommultiplikation nach Karatsuba mit M(N) = $N^{1,58}$

  (andere Möglichkeit: distributives Ausmultiplizieren)

  Divide: N = 1 $\Rightarrow$ berechne Produkt direkt

  sonst: p(x) = p_l(x) + p_r(x)$*x^{N/2}$

  q(x) = q_l(x) + q_r(x)$*x^{N/2}$

  p_m(x) = p_l(x) + p_r(x)

  q_m(x) = q_l(x) + q_r(x)

  Conquer: Rekursive Anwendung

  z_l = p_l(x) * q_l(x)

  z_r = p_r(x) * q_r(x)

  z_m = p_m(x) * q_m(x)

  Merge: Setze z = z_l(x) + $(z_m(x) - z_l(x) - z_r(x))*x^{N/2} + z_r(x)x^N$

  M(N) = M(2^k) = 3 M(2^{k-1}) = 3^k M(2^0) = $3^k = (2^{log3})^k = (2^k)^{log3} = N^{log3} = N^{1,58}$

- Rekursionsgleichungen:

  T(1) = b

  T(n) = a * T(n/c) + d*n

  $$T(n) \in \left\{ \begin{array}{ll} O(n), & \text{falls a} < \text{c} \\ O(nlog_c n) & \text{falls a} = \text{c} \\ O(n^{log_c a}) & \text{falls a} > \text{c} \end{array} \right\}$$

  T(1) = b

  T(n) = a * T(n/c) + d

  $$T(n) \in \left\{ \begin{array}{ll} O(log_c n), & \text{falls a} = 1 \\ O(n^{log_c a}) & \text{falls a} \neq 1 \end{array} \right\}$$

2 Sortieren

- Folge F von Datensätzen (items, records).
- Jeder Satz s_i hat Schlüssel k_i
- Ordnungsrelation:
  - natürliche Ordnung auf Zahlen
  - alphabetische und lexikographische Ordnung
- Messung der Laufzeit mit Anzahl der Schlüsselwertvergleiche(C Comparisons) und Bewegungen(M Movements)
- interne (vollständig im Hauptspeicher) und
  externe (externes Speichermedium) Sortierverfahren
- intern: Quicksort, da nicht viel Zwischenspeicher benötigt wird, nur eine konstante Anzahl an Hilfsspeicherplätzen
- extern: Mergesort, es wird viel Speicher benötigt

2.1 Mergesort - Sortieren durch Mischen

- Algorithmus

  {sortiert die Folge F durch rekursives Teilen nach aufsteigenden Werten}

  Falls F die Länge N = 0 oder N = 1 hat, so ist F bereits sortiert und bleibt unverändert. Sonst:

  Divide: teile in zwei etwa gleich große Teilfolgen F_1 und F_2

  Conquer: Sortiere F_1 und F_2 rekursiv mit Mergesort

  Mergesort(F_1) $\rightarrow$ F_1' sortiert

  Mergesort(F_2) $\rightarrow$ F_2' sortiert

  Merge: Bilde die Resultatfolge durch Verschmelzen von F_1' und F_2'. Lasse dazu je einen Positionszeiger (Index) durch die Teilfolgen F_1', F_2' so wandern: Bewege jeweils in einem Schritt denjenigen der beiden Zeiger um eine Position weiter, der auf den kleineren Schlüssel zeigt.

- Rekursionsgleichung:

  C(1) = 0, M(1) = 0

  C(N) = 2* C(N/2) + N, M(N) = 2 * M(N/2) + 2*N

- $C_{min}(N) = C_{max}(N) = C_{mit}(N) \in \Theta$(N * log N) (Schlüsselwertvergleiche)
- $M_{min}(N) = M_{max}(N) = M_{mit}(N) \in \Theta$(N * log N) (Bewegungen)

2.2 Quicksort

- ist erfahrungsgemäß (im Mittel) eines der schnellsten Sortierverfahren
- In-situ-Verfahren: kein zusätzlicher Speicher, außer konstante Anzahl von Hilfsspeicherplätzen
- Algorithmus

  { sortiert die Folge F durch rekursive Teilen nach aufsteigenden Werten }

  Falls F die Länge N = 0 oder N = 1 hat, so ist F bereits sortiert und bleibt unverändert. Sonst:

  Divide: Wähle ein Pivotelement k von F (z.B. das letzte) und teile F ohne k in Teilfolgen F_1 und F_2 bezüglich k: F_1 enthält nur die Elemente von F ohne k, die $\leq$ k sind, F_2 enthält die , die $\geq$ k sind

  Conquer: Sortiere F_1 und F_2 rekursiv mit Quicksort

  Quicksort(F_1) $\rightarrow$ F_1' sortiert

  Quicksort(F_2) $\rightarrow$ F_2' sortiert

  Merge: Bilde die Ergebnisfolge F' durch Hinterinanderhängen von F_1', k , F_2' in dieser Reihenfolge

- Aufwandsanalyse

  1. Im günstigsten Fall haben die durch Aufteilung entstehenden Teilfeolgen stets etwa die gleiche Länge. Dann ist die Rekursionstiefe genau von der Größenordnung log_2N. Da auf jeder Rekursionsstufe zur Aufteilung Θ(N) Schlüsselvergleiche durchgeführt werden gilt:
     $C_{min}(N) \in \Theta$(N * log N)
  2. ist Pivotelemnt das Element mit kleinsten oder größten Schlüssel
     $\Rightarrow C_{max}(N) \in \Omega(N^2)$ und $M_{max}(N) \in \Omega(N^2)$
  3. mittlerer Aufwand $C_{mit}(N) \in O$(N * log N) (Beweis per Induktion)
     zu N verschiedenen Schlüsseln ex. N! Anordnungen
     jedes Element ist als Pivotelement gleich wahrscheinlich
     T(1) = 0, T(N) $\leq$ 1/N $\sum_{k=1}^{N}(T(k-1)+T(N-k))+b*N$ (b*N Aufteilung)
     Integralabschätzung T(N) $\leq$ c * N * log N

- Worst-Case verbessern?

  Mit Mediansuche, verwende Median als Pivotelement, O(N * log N)

  Rekursionsformel T(N) $\leq$ T(N/2) + c*N (Mediansuche in c*N enthalten)

- Wie kann man aus Quicksort einen Baum gewinnen?

  Pivotelement jedes mal in den Knoten liefert AVL-Baum (mit Median)

2.3 Untere Schranken für das Sortierproblem

- Satz (Worst-Case)

  Jedes Sortierverfahren, das ausschließlich die vollständige Ordnung auf dem Wertebereich W ausnutzt benötigt im Worst-Case größenordnungsmäßig mindestens N * log_2 N Vergleiche zum Sortieren von N Werten

- Entscheidungsbaum

  ein bewerteter binärer Wurzelbaum B = (T,b) mit T = (V,R)

  - V: Knotenmenge von T
  - R $\subseteq$ V $\times$ V: Kantenmenge von T
  - b: R $\rightarrow \{ja, nein\}$: Kantenbewertung

  Jeder Knoten v $\in$ V der nicht Blatt ist repärsentiert eine Entscheidung (hier ein Vergleich w $<$ w')

  Jeder Pfeil r = (v,v') $\in$ R repräsentiert das Ergebnis einer Entscheidung an seinem Anfangsknoten

  Jedes Blatt v repräsentiert das Ergebnis des Entscheidungsprozesses längs des Weges von der Wurzel v_0 bis zu v , d.h. eine Partialordnung

  Jeder Knoten,der nicht Blatt ist hat einen oder zwei Nachfolger

  Jeder Sortieralgorithmus, der ausschließlich die vollständige Ordnung auf der Wertemenge W ausnutzt und sequentiell Wertepaare vergleicht, induziert einen solchen Entscheidungsbaum.

  Ein binärer Wurzelbaum der Höhe h_T hat maximal 2^{h_T} Blätter $\Rightarrow h_T \geq log_2$(N!)

  mit N! $\geq$ N/2 * (log_2N - 1) gilt $C_{max}(N) \in \Omega(N * log_2 N)$

- Satz (Average-Case)

  Jedes Sortierverfahren, das ausschließlich die vollständige Ordnung auf dem Wertebereich W ausnutzt benötigt im Mittel bei Gleichverteilungsannahme min. N*log_2N Vergleiche zum Sortieren von N Werten.

  Beweis: Entscheidungsbaum H(T) = $\sum$ h(v) ist Blätterhöhensumme

  zu zeigen: H(n) = min{H(T) | T ist binärer Wurzelbaum mit n Blättern}

  H(n) $\geq$ n * log_2 n (Beweis per Induktion nach n)

  Algorithmus der nur auf vollst. Ordnung der Zahlen beruhrt induziert Entscheidungsbaum mit Höhe h $\Rightarrow$ max 2^h Blätter

  $2^h \geq$ N!

  h $\geq$ log N! $\geq$ N/2 * (log N/2) = N/2 * (log N-1) $\Rightarrow C_{max} \in \Omega(Nlog_2N)$

- Beispiel:

2.4 Heaps und Heapsort

- Heap: Prioritätswarteschlange
- Heap

  Ein Feld a[1...N] mit Komponentenwerten a[i], $1 \leq i \leq N$ heißt Heap, falls gilt:

  1. a[i] $\leq$ a[2i], für $1 \leq i \leq \lfloor N/2 \rfloor$
  2. a[i] $\leq$ a[2i+1], für $1 \leq i \leq \lceil N/2 \rceil$
- Heap realisiert eine partielle Ordnung, nämlich einen vollständigen binären Wurzelbaum
- a[1] ist minimales Element des Heaps, Söhne sind immer größer (Partialordnung)
- Komplexität des Heap-Aufbaus

  Heap mit j Stufen speichert zwischen 2^{j-1} und $2^j - 1$ Schlüssl

  Auf Stufe k gibt es höchstens 2^{k-1} Schlüssel

  Anzahl der Bewege- und Vergleichsoperationen zum Versickern (downheap) $\in$ O(N)

  Heapsort: N-maliges Löschen in Log N: N * Log N
- Heapaufbau mit
  - upheap: Aufsteigen innerhalb der Liste $\in$ O(n)

    upheap(int k){

    while($(k > 1)$ && a[k] $<$ a[k/2]){

    exchange(k,k/2);

    k = k/2;

    }

    }

    $\Rightarrow$ Heapaufbau O(n log n) (kleine Elemente müssen durchgereicht werden)
  - downheap

    Prüfe ob a[k] linken(j = 2*k) und rechten Sohn hat, wähle kleineren davon

    Tausche mit k, falls a[k] $>$ a[j] (Sohn)

    Rekursiv weiter downheap mit Sohn (j)

    $\Rightarrow$ Heapaufbau $\in$ O(n), Prüfe nur Werte, die nicht in den Blättern stehen
  - Einfügen: als letztes Element in O(log N) mittels upheap
  - Entfernen: hole a[n] an Stelle a[k]

    upheap, falls a[k] $<$ a[k/2] oder downheap

	Aufbau Baum	Sortieren
Heapsort	N (downheap)	N * log N
Treesort	N * log N	N

2.5 Elementare Sortierverfahren

- Selection-Sort

  Man bestimme sukzessive für alle i diejenige Position j an der das Element mit minimalem Schlüsel unter den Elementen a[i],...,a[N] auftritt und vertauscht a[i] mit a[j]

  Komplexitäten $C_{mit} \in O(N^2)$, $M_{mit} \in$ O(N)

- Insertion-Sort

  Man füge skuzessive für alle i das i-te Feldelement a[i] an der richtigen Stelle in die bereits sortierte Folge der Elemente a[1],...,a[i-1] ein. Das verlangt das verschieben von j größeren Elementen um jeweils eine Position nach rechts

  $C_{min}(N)$ = N-1, $M_{min}(N)$ = 2*(N-1)

  $C_{max}(N) = M_{max}(N) \in \Theta(N^2)$

- Bubble-Sort

  Man durchläuft wiederholt die Liste der Datensätze a[1],...,a[i] und betrachtet dabei je zwei benachbarte Elemente a[j-1] und a[j]. Ist a[j-1] $>$ a[j], so vertauscht man a[j-1] und a[j]

  Große Elemnte haben also die Tendenz, wie Luftblasen im Wasser langsam nach oben aufzusteigen

  Komplexitäten $C_{mit} = M_{mit} \in \Theta(N^2)$

- Im schlechtesten Fall N^2 (elemantare Sortierverfahren)

- brechen aber rechtzeitig ab, wenn Folge schon sortiert

3 Dynamische Datenstrukturen

3.1 Lineare Listen

- Eine Folge L = $(a_1, ..., a_n)$ kann als lineare Liste von Knoten implementiert werden
- Ein Knoten besteht aus zwei Komponenten:
  - key-Komponente: Wert
  - next-Komponente: Verweis auf einen (anderen) Knoten (Zeiger)
- auch doppelt verkettete Speicherung möglich (next und prior)

3.2 Schlangen (queue)

- lineare Liste, aber Einfügen und Entfernen nur bei Listenanfang und -ende.
- Einfügen: pushhead(v), pushtail(v)
  Entfernen: v = pophead(), v = poptail()
  v = top() gibt das erste Element der Schlange zurück
  empty() testet ob Schlange leer
  init() erzeugt leere Schlange

3.3 Stapel (stack)

- lineare Liste, bei der das Einfügen und Entfernen eines Elements auf den Listenkopf beschränkt ist.
- Anwendung: Speicherung der Rücksprungadressen bei geschachtelten Programmaufrufen
- Sortieren mittels Insertion Sort, wobei 3 Stacks zur Verfügung: A,C zum Verschieben, B schon sortiert

3.4 Geordnete binäre Wurzelbäume

- Ein Wurzelbaum T = (V,E)
  - Menge V von Knoten
  - Menge E von Kanten E $\subseteq$ V $\times$ V

  so dass es einen eindeutigen Knoten (Wurzel) gibt, von dem alle Knoten über jeweils eindeutige Wege erreichbar sind
- Weg: Folge von Knoten, wobei für je zwei aufeinanderfolgende Knoten gilt $(v_i, v_{i+1}) \in$ E
- Vorgänger V(v) = $\{v' \in V | (v', v) \in E\}$
  Nachfolger N(v) = $\{v' \in V | (v, v') \in E\}$
- binärer Wurzelbam: jeder Knoten hat höchstens zwei Nachfolger
- geordneter binärer Wurzelbaum: für jeden Knoten v $\in$ V gibt es injektive Abbildung: O_v: N(v) $\rightarrow \{L, R\}$ (Nachfolgermenge)
  v′ $\in$ V mit O_v(v′) = L heißt linker Sohn
  v″ $\in$ V mit O_v(v″) = R heißt rechter Sohn
- Baumdarstellung mittels einer Zeigerstruktur
  Jeder Knoten hat Zeiger lSon und rSon (null, wenn nicht existent)
- Durchlauf

```
/**
* Generische Methode zum Baumdurchlauf entsprechend String id
* (ist von der Form "LWR", "WLR" oder "LRW")
*/
void durchlauf(String id, BinTreeNode t) {
        if (t != null) {
                for (int i = 0; i ¡ id.length(); i++) {
                        switch (id.charAt(i)) {
                                case 'L': durchlauf(id, t.lson); break;
                                case 'R': durchlauf(id, t.rson); break;
                                case 'W': System.out.println(t); break;
                        }
                }
        }
}
```

  $\Rightarrow$ kann zu linearer Liste degenerieren

- Ordnungen und Durchlaufprinzipien

  1. Inordnungen LWR,RWL Beim Durchlaufen wird zunächst
     - der linke bzw. rechte Teilbaum, dann
     - die Wurzel, und dann
     - der rechte bzw. linke Teilbaum

     durchlaufen
     (LWR und RWL sind stets invers zueinander)
  2. Randordnungen
     - Präordnungen (WLR,WRL)
       Hier wird die Wurzel vor den beiden Teilbäumen durchlaufen
     - Postordnungen (LRW,RLW)
       Hier werden die beiden Teilbäume vor der Wurzel durchlaufen
       WLR und RLW, sowie WRL und LRW sind zueinander invers.

- Lemma
  - Ein geordneter binärer Wurzelbaum ist eindeutig bestimmt durch die Angabe einer Inordnunge zusammen mit einer Randordnung
  - Die Angabe zweier Inordnungen bzw. Randordnungen reicht für die eindeutige Charakterisierung eines geordneten Wurzelbaums i.A. nicht aus

- $\Rightarrow$ LWR sortiert in O(N)

4 Suchen in Folgen

4.1 Sequentielle Suche

Man vergleicht der Reihe nach die Elemente $A_N, A_{N-1}, ..., A_1$ mit dem Suchschlüssel.

Um nicht immer prüfen zu müssen ob bereits alle Listenelemnte inspiziert wurden, verwendet man v selbst als Stopper an Position 0.

Worst-Case: $C_{max}(N)$ = N + 1

Average-Case: $C_{mit}(N)$ = (N+1)/2 $\in \Theta(N)$

4.2 Binäre Suche

Sucht in Liste von aufsteigend sortierten Schlüsseln nach einem Element a_m mit dem Suchschlüssel v:

Falls L leer ist, so endet die Suche erfolglos. Sonst betrachte das Element a_m an der mittleren Position m in L

(m = (N+1) /2)

- Falls v $< a_m$, so durchsuche die linke Teilliste $L_l = (a_1, ..., a_{m-1})$
- Falls v $> a_m$, so durchsuche die rechte Teilliste $L_r = (a_{m+1}, ..., a_N)$

Sonst ist a_m = v und die Suche war erfolgreich

- logarithmisch viele Schritte, weil sich bei jedem Schritt die Anzahl der Elemente halbiert
- Worst-Case und Average-Case $\in \Theta(log_2(N))$

4.3 Mediansuche

kleinstes Element in O(N) Schritten, i-kleinstes Element in O(i*N) Schritten
$\Rightarrow$ Median in O(N^2), mit Sortierverfahren: Median in O(N log N)

0. Rekursionsabbruch

   Falls N $\leq$ 91, so berechne das i-kleinste Element dirket, sonst:
   Schritt 1 bis 3: Bestimmung eines geeigneten Pivotelements v.

1. Aufteilung in Gruppen

   Teile N Elemnte in $\lfloor N/5 \rfloor$ Gruppen zu je 5 Elementen und höchstens eine Gruppe mit den restlichen (höchstens 4) Elementen auf

2. Bestimmung von Medianen

   Sortiere jede dieser höchstens $\lceil N/5 \rceil$ Gruppen aus maximal 4 oder 5 Elementen und greife das mittlere Element einer jeden 5er-Gruppe heraus. Für die letzte, kleiner Gruppe wähle man ebenfalls das mittlere Element, falls diese Gruppe eine ungerade Zahl von Elementen hat, sonst das größere der beiden mittleren Elemente. Man erhält somit $\lceil N/5 \rceil$ Mediane in der Zeit O(N)

   N/5 * 5 log 5 = N * log $\in$ O(N)

3. Rekursive Bestimmung des Medians der Mediane

   Wende das Auswahlverfahren für i = $\lceil \lceil N/5 \rceil /2 \rceil$ rekursiv auf die $\lceil N/5 \rceil$ Mediane an um das mittlere Element v dieser Mediane zu finden.

   v heißt Median der Mediane (liegt hinreichend Nahe am Median)

4. Aufteilung der Liste mit v als Pivotelement

   Teile die N Elemente bezüglich dem Pivotelement v auf in zwei Gruppen:
   Gruppe 1 enthält k Elemente, die kleiner als v sind, Gruppe 2 enthält N-k-1 Elemente, die größer als v sind. (Aufteilung in O(N))

   Beide Gruppe enthalten mindestens 3/10*N - 6 Elemente der Gesamtfolge L

5. Rekursive Anwendung des Auswahlverfahrens

   Falls i $\leq$ k ist, so liegt das i-kleinste Element von L in Gruppe 1 und wir suchen rekursive das i-kleinste Element der Gruppe 1.

   Falls i $>$ k+1 so liegt das i-kleinste-Element von L in Gruppe 2 und es kann rekursiv das (i-(k+1)-kleinste) Element aus Gruppe 2 bestimmt werden.

- Das i-kleinste Element in einer Folge von N paarweise verschiedenen Elemente kann in O(N) Schritten gefunden werden
- min g' = $\lfloor *(\lfloor N/5 \rfloor - 1) \rfloor$ volle 5er-Gruppen, deren mittleres Element größer als v

  g' $>$ 1/2 (N/5 - 2) - 1 = N/10 - 2

  jeweils 3 Elemente dieser Gruppe größer $\Rightarrow$ 3*N/10 - 6

5 Bäume

5.1 Suchbäume

T = (V,E) ein geordneter (Info ob rechter oder linker Nachfolger) binärer (0,1 oder 2 Nachfolger) Wurzelbaum.

T heißt sortiert: $\forall v \in V$:(v' $\in T_l$(v) $\Rightarrow$ s(v') $\leq$ s(v) $\wedge$ v'' $\in T_r$(v) $\Rightarrow$ s(v'') $\geq$ s(v)).

Falls T sortiert ist heißt T Suchbaum.

- Minimum: linkester Knoten, Maximum: rechtester Knoten
- in LWR-Ordnung durchläuft man die Schlüsselwerte monoton steigend
- Im Suchbaum alle Schlüssel paarweise verschieden
- Wörterbuchoperationen: Suchen, Einfügen, Löschen
  - Einfügen $\in$ O(log N)
    Einfügen von Schlüsselwert s: Falls s vorhanden wird s nicht eingefügt
    Andernfalls bricht die Suche erfolglos ab und man kann s als Sohn des erreichten Knotens einfügen
  - Löschen, 1. Version (kommutativ)
    a) v ist Blatt: v wird gelöscht
    b) v hat genau einen Sohn: v wird gelöscht und der Sohn von v wird zum Sohn von Vater(v)
    c) v hat zwei Söhne
    Seien v_l und v_r der linke bzw. rechte Sohn von v, und sei v_t der rechteste Knoten des linken Teilbaums von v
    Dann wird v im Baum gelöscht und v_l wird zum Sohn vom Vater(v). v_r wird zum rechten Sohn von v_t, d.h. der rechte Teilbum $T_r(v)$ wird in den $T_l(v)$ eingehängt
  - Löschen, 2.Version $\in$ O(log N)
    rechtester Sohn von v_t hat keinen rechten Sohn und ist geeigneter Seperator von $T_l(v)$ und $T_r(v)$
    v aus T löschen und durch v_t ersetzen, und v_t in $T_l(v)$ löschen
    Version 1 ist die Höhen h(T') häuft viel größer als die Höhe von T
    Version 2 h(T') $\leq$ h(T)
    Aufwand beim Löschen und bei Suchen in O(h)
  - Generieren (nicht in O(N) möglich)
    Aufwand best-case und average-case $\in \Theta(N * log_2 N)$
    Aufwand worst-case (lineare Liste) $\in \Theta(N^2)$
    Median-Suche verbessert auf O(N log N):
    Immer Median der noch einzufügenden Elemente einfügen
    Keine weitere Verbesserung, wg. Untere Schranke des Sortierproblems

5.2 AVL-Bäume (Adelson-Velski-Landis) (balancierte Suchbäume)

T = (V,E) geordneter binärer Wurzelbaum

- Balance eines Knoten $\beta(v)$ = h($T_r(v)$) - h($T_l(v)$),

  d.h. die Höhendifferenz zwischen dem rechten und dem linken Teilbeim von v, wobei h(T') = -1, falls T' leer

  Falls v Blatt β(v) = 0

  Falls v kein Blatt:

  - T_l(v) leer, so ist β(v) = h(T_r(v)) + 1
  - T_r(v) leer, so ist β(v) = -1 - h(T_l(v))
- AVL-Baum: $\forall$ v $\in$ V: $|\beta(v)| \leq 1$
- Aufbau AVL-Baum in O(N log N)
- Linksrotation von v, v' mit v = Vater(v')

  v' wird an die Stelle von v gezogen, v wird linker Sohn von v'

  linker Teilbaum T_l von v wird linker Teilbaum von v

  linker Teilbaum T_{r1} von v' wird rechter Teilbaum von v

  rechter Teilbaum T_{t2} von v' wird rechter Teilbaum von v'
- Rechtsrotation von v',v, v' = Vater(v)

  Umkehrung der Linksrotation

  Rotrechts(v',v) $\circ$ Rotlinks(v,v')(T) = T
- Links- bzw. Rechtsrotation ändert nicht LWR-Ordnung des Baumes
- Satz S = $\{s_1, ..., s_N\}$ Schlüsselwertmenge. Dann sind alle Suchbäume T' von S durch endlich viele Anwendungen von Rotlinks und Rotrechts aus T erzeugbar

  Beweis: aus linearer Liste kann jeder Baum erzeugt werden und umgekehrt
- für alle Transformationen braucht man Rotlinks und Rotrechts
- Definitionen

  i) Außengrad g^+(v) und Innengrad g^-(v) geben Anzahl der Nachvolger bzw. Vorgängen von v in T an
  ii) T heißt vollständig, falls jeder innner Knoten genau zwei Söhne hat
  iii) T heißt voll, falls alle Knoten auf allen Stufen außer den beiden untersten genau zwei Söhne haben
  iv) T voll $\Rightarrow$ T AVL-Baum
  v) T voll $\not\Leftarrow$ T vollständig
  vi) T vollständig $\not\Leftarrow$ T AVL-Baum

Fibonacci-Bäume:

- extremale AVL-Bäume, welche zu einer vorgegebenen Höhe h eine minimale Anzahl von Knoten aufweisen:

  n_h = min{n | T ist AVL-Baum mit n Knoten und Höhe h}

- Rekursionsformel zur Knotenminimalität

Knotenminimale AVL-Bäume	Fibonacci Zahlen
$n_0 = 1$	$f_0 = 0$
$n_1 = 2$	$f_1 = 1$
$n_h = 1 + n_{h-1} + n_{h-2}$	$f_h = f_{h-1} + f_{h-2}$

h	0	1	2	3	4	5	6	...
n_h	1	2	4	7	12	20	33	...
f_h	0	1	1	2	3	5	8	...

- Definiton (Fibonacci-Bäume, FB-Bäume)

  Die Menge F_h der FB-Bäume der Höhe h ist rekursiv definiert

  i) F_0 enthält alle Wurzelbäme T_0 = ({v},$\varnothing$) mit genau einem Knoten und Höhe h = 0

  ii) f_h, h ≥ 1 enthält alle geordneten Wurzelbäume $T_h = < T_{h-1}, v, T_{h-2} >$ bzw. $< T_{h-2}, v, T_{h-1} >$, wobei $T_{h-1} \in F_{h-1}$ und $T_{h-2} \in F_{h-2}$

- Satz: Jeder FB-Baum ist auch AVL-Baum

  Beweis: $T_h = < T_{h-1}, v, T_{h-2} >$ rekursiv $\Rightarrow |\beta(v)| \leq 1$

- Satz:

  Ein knotenminimaler AVL-Baum T zu einer vorgegebenen Höhe h mit n Knoten ist gleichzeitg auch ein höhenmaximaler AVL-Baum zur vorgegebenen Knotenzahl n

- Satz: Höhenabschätzung für AVL-Bäume (mit n Knoten)

  h(T) $\leq 2 * log_2 n$

- Es gibt $2^{n_{h-1}}$ FB-Bäume der Höhe h (strukturell gesehen)

- Doppelrotationen

  DoppelRotlinks(v,v',v'') = Rotlinks(v,v'') $\circ$ Rotrechts(v',v'')

  DoppelRotrechts(v,v',v'') = Rotrechts(v,v'') $\circ$ Rotlinks(v',v'')

- Rebalancierung

  T = $< T_1, v, T_2 >$ ein geordneter binärer Wurzelbaum mit Wurzel v und Teilbäumen T_l, T_r, welche beide AVL-Bäume sind.

  Ist $\beta(v)$ = 2, so kann T mittels einer Rotation oder Doppelrotation in AVL-Baum T' überführt werden

  Beweis:

  - linkslastig: β(v) = -2
    i) $\beta(v_l) \in \{-1, 0\}$, d.h. h(T_{l1}) $\geq$ h(T_{l2})
    Rotrechts(v,v_l)
    h(T') = h(T), falls $\beta(v_l)$ = 0
    h(T') = h(T) - 1, falls $\beta(v_l)$ = -1
    ii) $\beta(v_l)$ 1, d.h. h(T_{l1}) = h(T_{l2}) - 1
    DoppelRotrechts(v,v_l,v_{l2})
    h(T') = h(T) - 1

- Klassen balancierter Bäume

  f: $\mathbb{N} \rightarrow \mathbb{N}$ Funktion

  Klasse J von Wurzelbäumen balanciert zur Höhe f(n), g.d.w.

  1. Jede n-elementige Schlüsswertmenge S kann durch Baum der Höhe $h_T \leq$ f(n) repräsentiert werden
  2. Suchen, Einfügen, Löschen liefern neuen Baum und können in O(h_T) ausgeführt werden

- Klassen AVL ist zur Höhe f(n) = 2*log_2 n balanciert

  Rebalancierungen nach dem Adelson-Verfahren mit Rotationen und Doppelrotationen

5.3 B-Bäume

T = (V,R) ein geordneter Wurzelbaum mit Knotenmarkierung
s: V $\rightarrow 2^s$, welche jedem Knoten v einen Teilmenge s(v) $\subseteq$ S zuordnet
(innerer Knoten: nicht die Wurzel und kein Blatt)

i) Struktureigenschaften

a) Alle Blätter haben dieselbe Höhe h_T
b) Alle Knoten haben höchstes 2m + 1 Söhne. Die Wurzel hat min. 2 Söhne falls sie kein Blatt ist
c) Alle Knoten enthalten höchstens 2m Schlüssel
Die Wurzel enthält mindestens einen Schlüssel
alle anderen Knoten enthalten mindestens m Schlüssel
d) Hat ein Knoten k + 1 Söhne, k $\geq$ 0, so enthält er genau k Schlüssel,
D.h. g^+(v) = k+1 impliziert |s(v)| = k

ii) Suchbaumeigenschaft

Die Schlüsselwertmenge ist disjunkt auf die Knoten verteilt
s(v) = $\{s_1, ..., s_k\}$ Schlüssel $(v_1, ..., v_{k+1})$ Folge von Söhnen von v
Dann gilt:

a) Alle Schlüsselwerte s im ersten Teilbaum T(v_1) sind kleiner als s_1: s < s_1
b) im i-ten Teilbaum T(v_i) gilt: $s_{i-1} < s < s_i$
c) Alle Schlüsselwerte s im (k+1)-ten Teilbaum T(v_{k+1}) sind größer als s_k: s_k < s

- Wurzel: 1 $\leq$ k $\leq$ 2m
- andere Knoten: m $\leq$ k $\leq$ 2m
- alle Knoten außer Wurzel zu min 50 % mit Schlüsseln gefüllt
- B-Baum von Typ m ist zur Höhe f(n) = 1 / (log_2(m+1)) * log_2n balanciert

Wörterbuchoperationen

i) Suchen nach Schlüsselwert s: search(s,T)

Innerhalb Schlüsselwertmenge eines Knoten kann binär gesucht werden mit $\lceil log_2(2m) \rceil$
Weitere Suche

- s < s_1: Fortsetzung der Suche im 1.Teilbaum
- s_{i-1} < s < s_i: Fortsetzung der Suche im i-ten Teilbaum
- s_k < s: Fortsetzung der Suche im (k+1)ten Teilbaum

Aufwand in O(h_T)

ii) Einfügen eines Schlüsselwertes s: insert(s,T)

Suchen von s im Baum: ist s schon im Baum wird nicht angefügt, ansonsten Abbruch der Suche in einem Blatt v

s wird in das Blatt v eingefügt

Danach rebalanciern entlang des Pfades von der Wurzel zu v beginnend mit v:

Ein übergelaufener Knoten wird wiefolgt rebalanciert:

a) Die Schlüsselwerte in v^* werden in drei Gruppen aufgeteilt
   - die Menge S_1 der m kleinsten Schlüssel
   - das mittlere Element s_{m+1}
   - die Menge S_2 der m größten Schlüssel
b) Erzeugen von zwei neuen Knoten v_1 bzw. v_2 mit den Schlüsselwerten S_1 bzw. S_2
   Der Schlüssel s_{m+1} wird in den Vater von v^* eingefügt, falls v^* nicht die Wurzel des Baumes war. (sonst wird v^* neue Wurzel)
   rekursive Rebalancierung

Aufwand für das Einfügen: O(h_T), weil höchstens h_T + 1 mal gesplittet werden kann

Alternativ kann man beim Einfügen auch nach links (rechts) verschieben

iii) Löschen eines Schlüsselwerts s: delete(s,T)

Pfad von der Wurzel zu Knoten v, der s enthält. s ist i-ter Schlüssel in v

v Blatt: s wird aus v gelöscht

v kein Blatt: s wird aus v gelöscht und neuer Seperator s'. Dazu wählen wir den größten Schlüssel s' im rechtesten Blatt v' des i-ten Teilbaums T(v_i) von v

Wir löschen s' aus v' und ziehen s' nach v hoch

Wiederherstellen der Struktureigenschaft(Rebalancieren)

a) Falls v^* min. m Schlüssel enthält oder die Wurzel des Baumes ist, ist nichts zu tun
b) Falls v^* nur m-1 Schlüssel enthält (Unterlauf), so betrachten wir den linken und den rechten Bruder von v^*
   Enthält einer der Beiden Brüder mindestens m+1 Schlüssel, so kann man einen dieser Schlüssel verschieben.
   Enthalten beide Brüder nur m Schlüssel, so kann man v^* mit einen Bruder verschmelzen.
   Danach müssen wir evtl. den Knoten Vater(v^*) rebalancieren

Aufwand: Hochziehen und Verschieben wird höchstens einmal ausgeführt, Verketten bei der Rebalanicerung höchstens h_T mal

Löschen in O(h_T)

- B-Bäume sind die optimalsten (a,b)-Trees, in denen gilt:
  - 2*(a-1) $\leq$ b-1 (Anzahl Nachfolger eines Knotens)
  - a′ = a-1, b′ = b-1 (Anzahl Schlüssel eines Knotens)
  - 2*a′ $\leq$ b′

  B-Bäume mit a′ = m, b′ = 2m
- Bei praktischen Problemstellugne sind die Höhen der B-Bäume gewöhnlich höchstens 3. (m=50)
- Höhenvergleich:
  - B-Bäume: $\frac{log_2(n+1)}{log_2(2m+1} - 1 \leq h \leq \frac{log_2(n+1)-1}{log_2(m+1)}$
  - AVL-Bäume: $log_2(n+1) - 1 \leq h \leq 1,44 * (log_2(n+2) + \frac{1}{2} * log_2(5)) - 3$
- B^*-Bäume (blätterorientierte Version der B-Bäume)
  - Nicht-Blatt-Knoten dienen - wie üblich - als Wegweiser zu den Schlüsselwerten
  - Datensätze werden entsprechend ihrer Schlüsselwerte aus S diskjunkt auf die Blätter verteilt, welche zwischen m' und 2m' Datensätze enthalten
  - Separatoren separieren mit $\leq$ anstatt $<$
- Auf Blätterebenen bereits die gesamte Information mit vollständigen Datensätzen
- Anwendung: zu den Schlüsseln s_i auch beschreibende Attribute mit viel Speicherverbrauch
- Bei Suche in B-Baum muss man in etwa genauso viele Schlüssel anschauen, wie in AVL-Baum

  (AVL: mehr Knoten, B: richtige Lücke)
- B-Bäume: weniger Seitenwechsel

  bei großen Schlüsselwertmengen ist auch Laden der Seiten entscheidend
- Schneller als B-Bäume $\Rightarrow$ Hashverfahren

6 Hashverfahren

6.1 Grundbegriffe

- universelle Schlüsselmenge S $\subseteq \mathbb{N}$, Schlüsselwerte s $\in \mathbb{N}$
- Zuordnung von s auf Speicherplatz a $\in$ A erfolgt mittels arithmetischer Funktion f: S $\rightarrow$ A,

  genannt Hashfunktion: a = f(s) heißt Hauptadresse von s
- bei B-Bäumen: datenorientierte Strukturierung

  bei Hashverfahren: speicherorientiert
- ist f injektiv, so heißt f auch direkt

  andernfalls heißt f eigentlich Hashfunktion

  - S = $< 1, N > \subseteq \mathbb{N}$
  - A = $< 1, M > \subseteq \mathbb{N}$

  Meist ist M $<$ N und f eigentliche Hashfunktion
- Beispiele

  i) N = 30, M = 70; $f_1(s)$ = 6 + 2*s (direkt)

  ii) N = 30, M = 11; $f_2(s)$ = 1 + $\lfloor s/3 \rfloor$ (eigentlich)

s	1	2	3	4	5	6	...	27	28	29	30
$f_1(s)$	8	10	12	14	16	18	...	60	62	64	66
$f_2(s)$	1	1	2	2	2	3	...	10	10	10	11

- Synonyme: s, s', falls sie dieselbe Hauptadresse f(s) = f(s') besitzen

  Äquivalenzklassen $S_i \subseteq$ S mit $\forall s \in S_i$: f(s) = i heißen Synonymklassen
- Direkte Speicherfunktionen führe oft zu Speicherplatzverschwendung

  $\Rightarrow$ sehr geringer Belegungsfaktor $\alpha = \frac{|S|}{|A|} << 1$
- Hashverfahren besteht aus

  - (eigentliche) Hashfunktion (einfach zu berechnene)
  - Kollisionsstrategie (aufwendiger)

6.2 Gebräuchliche Hashfunktionen

i) Division mit Rest

A = $< 1, M >$

f(s) = b + s mod M

mit Basisadresse b $\in \mathbb{N}_+$

Ist $|A| = 10^k$, so entspricht Modulu-Berechnung dem Abscheiden der letzten k Stellen von s (Sectioning) (aber hohe Kollisionsgefahr)

Wähle für Größe M eine Primzahl

ii) Basistransformation

s = $s_0 + s_1 * 10 + s_2 * 10^2 + ... + s_k * 10^k = \sum_{i=0}^{k} (s_i * 10^i)$

Wähle neue Basis: f(s) = $\sum_{i=0}^{k} (s_i * b^i)$

Kombination der Basistransformation mit der Division mit Rest:

f(s) = 1 + $(\sum_{i=0}^{k} (s_i * b^i))$ mod M

iii) Multiplikationsmethode

reelle Zahl b > 0

f(s) $\lfloor (s * b - \lfloor s * b \rfloor) * M \rfloor + 1$

$(s * b - \lfloor s * b \rfloor) \in [0, 1]$ ist gebrochener Anteil von s*b

gleichmäßige Verteilung in $[0, 1]$ falls b irrational

b = $(\sqrt{5} - 1)/2 \approx 0{,}618$ (goldener Schnitt $\Rightarrow$ Fibonacci-Hashing)

$\Rightarrow$ gleichmäßige Verteilung

6.3 Kollisionsstrategien

i) Überlaufhash

gespeicherte Synonyme zur Hausadresse im Überlaufbereich (lineare Liste)

worst-case C(n) = $\Theta(n)$

- Suchen: Hausadresse berechnen, suche s in a und Überlaufbereich
- Einfügen: Hausadresse leer: speichern, ansonsten suchen und im Überlaufbereich einfügen
- Löschen: Suchen des Schlüssels: falls auf Hausadresse: ersetzen durch synonymen Schlüssel und löschen

Aufwand

- erfolgreiche Suche C(n) $\leq 1 + \alpha/4$*(n+1)
- erfolglose Such C(n) $\leq 1 + \alpha + \alpha^2/2$

ii) Überlauf mit Verkettung (Überlauf wie oben)

jede Adresse a wird durch Hashfunktion mit der gleichen Belegungswahrscheinlichkeit p(a) belegt: p(a) = $1/|A|$ = 1/M

Erwartungswert ξ für Länge des Überlaufbereichs

$\xi = \sum \xi_i * p(i) = 1/M * \sum \xi_i = n/M = \alpha$

Aufwand

- erfolgreiche Suche C(n) = $(1 + \alpha)/2$, mit $|S| < |A|$ gilt C(n) $\in$ O(1)
- erfolglose Suche C(n) = $\xi = \alpha$, mit $|S| < |A|$ gilt C(n) $\in$ O(1)
- worst: Θ(n)

iii) Offener Hash (Open Adressing)

für jedes s wird Permutation $\Pi(s) = (a_{i1}, ..., a_{iM})$ aller Hausadressen angegeben

s wird unter erster freier Adresse a_{ij} abgelegt

a_{i1} ist Hausadresse, a_{i2},...,a_{iM} sind Ausweichadressen

a) Lineares Suchen (Linear Probing)

$\Pi_+(s)$ = (f(s),f(s)+1,...,M-1,M,0,1,...,f(s)-1)

einfach, aber hat Nachteile: primary Clustering (Lücken zwischen längeren Teilstück werden geschlossen, sobald α gegen 1 geht)

Average-Case

- erfolgreiche Suche C(n) $\approx$ (1 + 1/(1-α)) $< 3/2$, für $\alpha < 1/2$
- erfolglose Suche C(n) $\approx$ (1 + $1/(1-\alpha)^2$) $< 5/2$, für $\alpha < 1/2$

b) Doppel-Hash (Double Hashing)

Vermeiden von primärem Clustering durch zweite Hashfunktion (Inkrementfunktion) f': S $\rightarrow$ A

a_{ij} = 1 + (f(s) - 1 + (j-1)*f'(s)) mod M, $1 \leq$ j $\leq$ M

Man fordert ggT(f'(s),M) = 1 (teilerfremd) (Primzahlenzwillinge)

Suchzeit hängt von der Reihenfolge des Ladens der Schlüsselwerte ab