
Effiziente Algorithmen und Datenstrukturen

Dr. S. Kosub

Erstellt von Benjamin Gufler

Inhaltsverzeichnis

1	Konzepte und Methoden der Algorithmenanalyse	1
1.1	Theorie und Experiment	1
1.2	Komplexitätsmaße und -arten	2
1.2.1	Amortisationsanalyse	4
1.3	Analyse randomisierter Algorithmen	7
2	Grundlegende Datenstrukturen	11
2.1	Stacks und Queues	11
2.2	Bäume	11
2.3	Priority Queues und Heaps	13
2.4	Wörterbuch und Hashing	16
2.4.1	Strategien zur Kollisionauflösung	17
	Geschlossene Hashverfahren	17
	Offene Hashverfahren	17
	Universelles Hashing	19
3	Suchbäume und Skiplisten	21
3.1	Binäre Suchbäume	21
3.2	AVL – Bäume	23
3.3	(a, b) – Bäume	26
3.4	Splay Trees	28
3.5	Skiplisten	32
4	Sortieren und selektieren in Mengen	35
4.1	Analyse von Quicksort	35
4.2	Selektionsalgorithmen	39
4.3	Höhere Heapstrukturen: Binomial Queues	41
4.4	Höhere Heapstrukturen mit Fibonacci-Heaps	43
4.5	Höhere Heap-Strukturen: Soft Heaps	45
4.6	Mengen und Union-Find-Strukturen	51
5	Graphenalgorithmien	55
5.1	Kürzeste Wege	55

Kapitel 1

Konzepte und Methoden der Algorithmenanalyse

Ziel der Vorlesung: Kennenlernen der Entwurfsprinzipien »guter« Algorithmen und Datenstrukturen.

Algorithmus: schrittweises Verfahren zur Lösung bestimmter Aufgaben (typischerweise in endlicher Zeit und endlichem Platz)

Datenstrukturen: Systematik zur Organisation und Verwaltung von Daten

»gut«: korrekt und möglichst geringe Laufzeit (geringer Speicherverbrauch)

1.1 Theorie und Experiment

Arten der Algorithmenanalyse:

- ➔ experimentelle Analyse
- ➔ theoretische oder asymptotische Analyse

1. Experimentelle Analyse

Bestimme den Zusammenhang zwischen Eingabegrößen und Laufzeit (typischerweise: Millisekunden etc.) eines implementierten Programmes auf einer Menge von Testdaten.

Typisches Resultat: Diagramm (Punktwolke; ein Punkt (n, t) bedeutet: bei *einer* Eingabe der Länge n hat der Algorithmus t ms gebraucht).

Nachteile:

- Experimente nur auf beschränkter Testmenge
- Algorithmus muss implementiert und ausgeführt werden
- Testmenge muss nicht repräsentativ für das Problem sein
- Algorithmen nur vergleichbar bei Ausführung in gleicher Hardware- und Softwareumgebung

2. Theoretische (oder asymptotische) Analyse

Bestimme den Zusammenhang zwischen Eingabegrößen und Laufzeit (typischerweise: Anzahl der Schritte etc.) auf mathematisch – deduktiver Basis.

Anforderungen:

- Betrachtung aller möglichen Eingaben
- relative Effizienz zweier Algorithmen unabhängig von Hardware- und Softwareumgebung analysieren

- Analyse möglich auf Basis von High – Level – Beschreibung von Algorithmen (ohne Implementierung)

Typisches Resultat: Funktion (z.B. »Algorithmus A arbeitet in zu n proportionaler Zeit«) Komponenten für eine theoretische Analyse:

- Beschreibungssprache (Pseudo – Code, ...)
- Berechnungsmodell (Registermaschine, Turingmaschine, ...)
- Laufzeitmaße (Bitkomplexität, worst case, ...)
- mathematische Einbettung der Laufzeitfunktionen (Asymptotik: $O, o, \omega, \Omega, \Theta$, Auflösung von Rekursion)

Nachteile:

- Algorithmus kann zu kompliziert sein, um aussagekräftige mathematische Analyse zu erlauben
- Asymptotik gibt keine Antwort über Konstanten
- keine Auskunft, ab wann ein asymptotisch schneller Algorithmus mit großer Konstante *besser* ist als ein asymptotisch langsamer Algorithmus mit kleiner Konstante

1.2 Komplexitätsmaße und -arten

Eine Laufzeitfunktion stellt eine Beziehung zwischen Eingabegröße und Laufzeit her.

1. Was ist die Eingabegröße?

- uniforme Eingabegröße: Anzahl der Komponenten, aus denen die Eingabe besteht; z.B.: die uniforme Größe von $x = (x_1, \dots, x_m)$ ist m .
- logarithmische Eingabegröße: Länge der Binärdarstellung der Eingabe (Zahl); es gilt: $|\text{bin}(n)| = \lfloor \log(n) \rfloor + 1$

2. Was ist die Laufzeit?

- Anzahl primitiver Operationen (im RAM)
 - Variablenzuweisung: 1 Schritt
 - Methodenaufruf: 1 Schritt
 - arithmetische Operationen: ?
 - Vergleiche: 1 Schritt
 - Feldindizierung: 1 Schritt
 - Rückkehr aus Methode: 1 Schritt
- uniformes Komplexitätsmaß: arithmetische Operationen in konstanter Zeit unabhängig von der Größe der Zahlen
- logarithmische Komplexität (Bitkomplexität): Kosten arithmetischer Operationen müssen auf Bitebene ausgerechnet werden; z.B. zwei n – Bit – Zahlen in Zeit $O(n)$ addierbar, in $O(n^2)$ multiplizierbar

Beispiel: Bestimmung des Maximums

Betrachte folgenden Algorithmus auf Eingabe in Array $A[1..n]$ mit $n \geq 1$, n ist Eingabelänge.

```

1  max := A[1];
2  for i := 2 to n
3    if A[i] > max
4      max := A[i];

```

3. Welche Art Beziehung zwischen Eingabe und Laufzeit?

Sei $\text{time}_A(x)$ die Anzahl primitiver Operationen, die von Algorithmus A auf Eingabe x ausgeführt werden.

(a) worst – case – Komplexität:

$$\text{wc-time}_A(n) =_{\text{def}} \max_{|x|=n} \text{time}_A(x)$$

Interpretation:

- $\text{wc-time}_A(n) \leq f(n) \Rightarrow A$ führt auf *jeder* Eingabe der Länge n höchstens $f(n)$ Schritte aus.
- $\text{wc-time}_A(n) \geq g(n) \Rightarrow$ es gibt eine Eingabe der Länge n , auf der A mindestens $g(n)$ Schritte ausführt.

Im Beispiel:

$$\begin{aligned} \text{wc-time}_A(n) &= \underbrace{2}_{(1)} + (n-1) \left(\underbrace{2}_{(2)} + \underbrace{2}_{(3)} + \underbrace{2}_{(4)} + \underbrace{1}_{\text{Inkr. i}} \right) + 2 \\ &= 7n - 3 \end{aligned}$$

(b) best – case – Komplexität:

$$\text{bc-time}_A(n) =_{\text{def}} \min_{|x|=n} \text{time}_A(x)$$

Interpretation:

- $\text{bc-time}_A(n) \leq f(n) \Rightarrow$ es gibt eine Eingabe der Länge n , auf der A höchstens $f(n)$ Schritte ausführt
- $\text{bc-time}_A(n) \geq g(n) \Rightarrow A$ führt auf *jeder* Eingabe der Länge n mindestens $g(n)$ Schritte aus

Im Beispiel:

$$\begin{aligned} \text{bc-time}_A(n) &= \underbrace{2}_{(1)} + (n-1) \left(\underbrace{2}_{(2)} + \underbrace{2}_{(3)} + \underbrace{1}_{\text{Inkr. i}} \right) + 2 \\ &= 5n - 1 \end{aligned}$$

(c) average – case – Komplexität:

$$\text{av-time}_A(n) =_{\text{def}} \mathbb{E}T_{A,n}$$

wobei $T_{A,n}$ Zufallsvariable sei, die die Laufzeit von A bei zufälliger Wahl der Eingabe der Länge n angibt; generell wird Gleichverteilung auf Eingaben der Länge n angenommen. Mit anderen Worten:

$$\text{av-time}_A(n) =_{\text{def}} \frac{1}{|\{x : |x|=n\}|} \sum_{|x|=n} \text{time}_A(x)$$

Im Beispiel:

Wir haben ein Feld $\mathbf{A}[1..n]$ von n verschiedenen Zahlen (z.B. $\{1 \dots n\}$) gegeben, d.h. eine beliebige Permutation von n Zahlen. Dann gilt:

$$\begin{aligned} \mathbb{P}(A[2] > A[1]) &= \frac{1}{2} \\ \mathbb{P}(A[3] > \max\{A[1], A[2]\}) &= \frac{1}{3} \\ &\vdots \\ \mathbb{P}(A[n] > \max\{A[1], \dots, A[n-1]\}) &= \frac{(n-1)!}{n!} = \frac{1}{n} \end{aligned}$$

Es sei x_j für $j \in \{1, \dots, n\}$ Zufallsvariable mit

$$x_j = \begin{cases} 1 & \text{falls } A[j] \text{ in } A[1..j] \text{ maximal ist} \\ 0 & \text{sonst} \end{cases}$$

Dann ist $\mathbb{E}x_j = \mathbb{P}(A[j] > \max\{A[1], \dots, A[j-1]\}) = \frac{1}{j}$. Mit $T_{A,n} = 2 + (n-1)(2+2+1) + 2 + 2 \sum_{j=2}^n x_j$ gilt

$$\begin{aligned} \text{av-time}_A(n) &= \mathbb{E}T_{A,n} \\ &= 5n - 1 + 2\mathbb{E} \sum_{j=2}^n x_j \\ &= 5n - 1 + 2 \sum_{j=2}^n \mathbb{E}x_j \\ &= 5n - 1 + 2 \sum_{j=2}^n \frac{1}{j} \\ &= 5n - 1 + 2(H_n - 1) \\ &\approx 5n - 1 + 2(\ln n - 1) \end{aligned}$$

(wobei H_n die harmonische Reihe darstellt; es gilt:

- $\ln(n+1) \leq H_n \leq \ln n + 1$
- $H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon$ mit $0 < \varepsilon < \frac{1}{252n^6}$, $\gamma = 0,5772156649\dots$ (eulersche Konstante)

)

Zusammenfassung:

- best-case: $5n - 1$
- average-case: $5n + 2 \ln n - 1$
- worst-case: $7n - 1$

(immer $\Theta(n)$)

Frage: Gibt es einen schnelleren Algorithmus? Insbesondere: Gibt es einen Algorithmus, der mit weniger als $n - 1$ Vergleichen auskommt?

1.2.1 Amortisationsanalyse

Typische Situation bei Datenstrukturen: Es wird eine Folge von Grundoperationen (Einfügen, Löschen, Inkrementieren) ausgeführt. Wird für jede Grundoperation der worst - case betrachtet, so ergibt sich im Allgemeinen eine zu pessimistische Laufzeit.

Amortisationsanalyse studiert worst - case - Gesamtkomplexität bei der sequentiellen Ausführung von n Grundoperationen.

Sei $T(n)$ die Gesamtkomplexität. Dann ist die AMORTISIERTE KOMPLEXITÄT der Einzeloperation definiert als $\frac{T(n)}{n}$.

Bemerkungen:

- Amortisierte Komplexität ist von average - case - Komplexität zu unterscheiden:
 - average - case: Wahrscheinlichkeitsverteilung über Einzeleingaben
 - Amortisation: durchschnittliche Komplexität in *einer* Folge von Operationen

Varianten für Amortisationsanalyse:

1. Aggregatmethode
2. Bankkontomethode
3. Potenzialmethode

Beispiel: Binärzähler

Grundoperation ist Inkrementierung eines Zählers, beginnend bei 0. Komplexität sei die Anzahl der geänderten Bits. Insgesamt soll n Mal inkrementiert werden.

worst – case – Abschätzung der Gesamtkomplexität: $O(n \log n)$

Amortisationsanalyse:

1. AGGREGATMETHODE:

Allgemein: schätze Gesamtkomplexität direkt ab.

Im Beispiel:

Zähler	Komplexität
0000	-
0001	1
0010	2
0011	1
0100	3
0101	1
0110	2
0111	1
1000	4
1001	1
...	...

d.h.:

- bei jedem Zählvorgang entsteht Komplexität 1
- bei jedem zweiten Zählvorgang entsteht zusätzlich Komplexität 1
- bei jedem vierten Zählvorgang entsteht zusätzlich Komplexität 1
- ...

Insgesamt:

$$T(n) \leq n + \frac{n}{2} + \frac{2}{4} + \dots \leq \sum_{i=0}^{\infty} 2^{-i} n = 2n$$

d.h. amortisierte Komplexität des Inkrementierens in einem Binärzähler: ≤ 2 .

2. BANKKONTOMETHODE:

Allgemein: Seien t_1, t_2, \dots, t_n die Laufzeiten der Einzeloperationen. Interpretiere t_i als Gebühr, die an eine »Komplexitätskasse« entrichtet werden muss, bevor die i -te Operation ausgeführt werden darf.

Annahme: Vor jeder Operation erhält der Algorithmus ein »Gehalt« G .

Alle Operationen müssen aus den regelmäßig eingehenden Gehaltsbeträgen bezahlt werden (d.h. sparen ist möglich, aber es gibt keine Kredite). Gibt es einen geeigneten Gehaltsbetrag G und eine geeignete Sparstrategie, so gilt

$$T(n) = \sum_{i=1}^n t_i \leq nG$$

damit: amortisierte Komplexität $\leq G$.

Im Beispiel:

- $G = 2$
- Sparstrategie:
 - Bei jedem Zählvorgang wird genau ein Bit auf 1 gesetzt; dies kostet eine Einheit.
 - Die zweite Einheit wird gespart, um zu einem späteren Zeitpunkt dieses Bit wieder auf 0 zurückzusetzen.

Damit: Zum Zeitpunkt i fällt k Mal Rücksetzen auf 0 und einmal Bitsetzen an, dies kostet $k + 1$ Einheiten. Finanziert wird dies durch k Einheiten aus dem »Sparkonto« und einer Einheit aus dem Gehaltseingang.

3. POTENZIALMETHODE:

Allgemein: Operationen können potenzielle Energie erzeugen und verbrauchen. Amortisierte Komplexität der i -ten Grundoperation:

$$a_i = t_i + \Delta\Phi = t_i + \Phi_i - \Phi_{i-1} \quad (\Phi \text{ heißt Potenzial})$$

Die Gesamtkosten ergeben sich dann zu

$$T(n) = \sum_{i=1}^n t_i = \sum_{i=1}^n (a_i + \Phi_{i-1} - \Phi_i) = \left(\sum_{i=1}^n a_i \right) + \Phi_0 - \Phi_n$$

D.h. falls $\Phi_n \geq \Phi_0$ ist, sind die amortisierten Kosten eine obere Schranke für die Gesamtkosten.

Problem: Bestimme ein geeignetes Potenzial.

Im Beispiel:

- Definiere $\Phi_i = |\text{bin}(i)|_1$ (Anzahl der Einsen in der Binärdarstellung von i).
- Klar ist: $\Phi_i \geq \Phi_0 \forall i \geq 0$.
- Betrachte den Übergang von $i-1$ zu i . Dabei werden k_i Bits auf 0 gesetzt und ein Bit von 0 auf 1: $t_i = k_i + 1$.
- $\Phi_i - \Phi_{i-1} = \Phi_{i-1} - k_i + 1 - \Phi_{i-1} = 1 - k_i$
- Damit ergibt sich insgesamt:

$$a_i = t_i + \Phi_i - \Phi_{i-1} = (k_i + 1) + (1 - k_i) = 2$$

Beispiel: Dynamische Arrays

- Typisches Problem bei Verwendung von Arrays: Größe N des Arrays muss deklariert werden (und ist damit fest).
- Falls die Anzahl zu speichernder Einträge $n < N$: Verschwendung von Speicherplatz.
- Falls $n > N$: Array zu klein.
- Lösung: Dynamische Allokation von Speicherplatz.
- Kritische Operation: Füge ein neues Element in ein Array A bei $n = N$ ein.

Verfahre wie folgt:

- Initialisiere neues Array B der Größe $2N$.
- Kopiere A nach B , d.h. `for i := 1 to N do B[i] := A[i]`.
- Setze A auf B .

Komplexität der Speicherung einer Tabelle S der Größe n :

- im schlechtesten Fall dauert das Einfügen $\Theta(n)$.
- D.h. insgesamt $O(n^2)$?

Satz 1 Es sei S eine Tabelle, die als dynamisches Array A (mit multiplikativer Vergrößerungsstrategie) implementiert ist. Die Laufzeit für das Ausführen von n Einfügeoperationen in S für den Fall, dass am Anfang S leer ist und $N = 1$ für die Größe von A gilt, ist $O(n)$.

Beweis. Wir verwenden die Bankkontomethode der amortisierten Analyse. Die Komplexität der Einfügeoperation (ohne Vergrößerung) sei 1. Die Komplexität für die Vergrößerung von k auf $2k$ seien k Schritte (für das Kopieren). Wähle $G = 3$. Zu zeigen: Es gibt eine geeignete Sparstrategie bei diesem Gehalt.

- Für jedes Einfügen bezahle 1 Einheit.

- Vergrößerung nötig, falls S genau 2^i Einträge enthält; Verdoppelung der Arraygröße auf $2 \cdot 2^i = 2^{i+1}$ kostet 2^i Einheiten.
- Sparvolumen: gesparte Einheiten seit letzter Verdoppelung: $2(2^i - 2^{i-1}) = 2^i$

$$\Rightarrow T(n) \leq 3n = O(n)$$

□

Frage: Wie sieht es aus mit additiver Vergrößerung?

Satz 2 Es sei $L > 0$ eine beliebige additive Vergrößerungskonstante. Sei S eine Tabelle, die als dynamisches Array A mit additiver Vergrößerungsstrategie implementiert ist. Die Laufzeit für das Ausführen von n Einfügeoperationen ist $\Omega(n^2)$.

Beweis. Übungsaufgabe

□

1.3 Analyse randomisierter Algorithmen

Randomisierte (probabilistische, stochastische) Algorithmen verwenden zur Steuerung des Programmablaufs Zufallszahlen. Damit sind sowohl Laufzeit als auch Ergebnis Zufallsvariablen (in Abhängigkeit von der Eingabe). Wir setzen dafür einen idealen Zufallsgenerator voraus. In Pseudocode: `RANDOM x IN M` mit Interpretation: x zufällig gleichverteilt (und unabhängig) aus der endlichen Menge M gezogen.

Beispiel: Randomisiertes Bubblesort

Eingabe: Array $[1 \dots n]$ mit natürlichen Zahlen.

```

1 REPEAT
2   RANDOM i IN {1, ..., n-1}
3   IF A[i] > A[i+1]
4     Vertausche( A[i], A[i+1] )
5 UNTIL ( A ist sortiert )
```

Beachte: Nur die Laufzeit ist Zufallsvariable!

Zufallsarten

Definiere:

$$\text{rtime}_A(x) =_{\text{def}} \mathbb{E} \text{time}_A(x)$$

wobei der Erwartungswert von $\text{time}_A(x)$ über alle im Programmablauf von A möglichen Zufallsauswahlen ermittelt wird (nicht über die Eingaben!).

- worst – case:

$$\text{wc-time}_A(n) = \max_{|x|=n} \text{rtime}_A(x)$$

- average – case:

$$\text{av-time}_A(n) = \frac{1}{|\{x : |x| = n\}|} \sum_{|x|=n} \text{rtime}_A(x)$$

- best – case:

$$\text{bc-time}_A(n) = \min_{|x|=n} \text{rtime}_A(x)$$

Beispiel: Randomisierter Gleichheitstest

Gegeben seien Arrays $A[1..n]$ und $B[1..n]$. Frage: $A \equiv B$? (d.h. sind die Zahlen in A und B als Multimenge gleich?)

Deterministischer Ansatz: Sortiere A und B und vergleiche dann. Komplexität $O(n \log n)$.

Randomisierter Ansatz liefert $O(n)$!

Idee: Ordne A und B Polynome p_A und p_B zu:

$$p_A(x) =_{\text{def}} \prod_{i=1}^n (x - A[i])$$

$$p_B(x) =_{\text{def}} \prod_{i=1}^n (x - B[i])$$

Setze $q(x) = p_A(x) - p_B(x)$. Es gilt:

- $\deg p_A = \deg p_B = n$
- $\deg q \leq n$
- $A \equiv B \Leftrightarrow q \equiv 0$ (Nullpolynom)

Sei S eine endliche Menge mit $\|S\| > n$. S ist die Menge von Nullstellenkandidaten. Dann gilt:

- $A \equiv B \Rightarrow \mathbb{P}_S(q(x) = 0) = 1$
- $A \not\equiv B \Rightarrow \mathbb{P}_S(q(x) = 0) \leq \frac{n}{\|S\|}$

Betrachte dazu folgenden Algorithmus (**epsilon** Fehlerwahrscheinlichkeit):

```

1  RANDOM x IN {1, 2, ..., ceil( n/epsilon )};
2  a := 1;
3  b := 1;
4  FOR i := 1 TO n
5    a := a * ( x - A[i] );
6    b := b * ( x - B[i] );
7  IF a = b
8    RETURN "A und B sind gleich"
9  ELSE
10   RETURN "A und B sind nicht gleich"
```

Es gilt:

- Laufzeit ist für Komplexitätsarten $\Theta(n)$ im uniformen Komplexitätsmaß
- Zufallszahl wirkt sich nur auf Ergebnis aus
- Algorithmus macht nur einseitigen Fehler:
 - $A \equiv B \Rightarrow$ Algorithmus antwortet immer korrekt
 - $A \not\equiv B \Rightarrow$ Algorithmus antwortet mit Wahrscheinlichkeit $\leq \text{epsilon}$ falsch bzw. mit Wahrscheinlichkeit $\geq 1 - \text{epsilon}$ korrekt.

Methoden zur Herabsetzung der Fehlerwahrscheinlichkeit

- Vergrößere Menge S ; aber nur lineare Verringerung der Fehlerwahrscheinlichkeit.
- Amplifikation mit exponentieller Verringerung.

Amplifikation mit beidseitigem Fehler

Angenommen, wir haben einen Algorithmus **ALG**, der den randomisierten Gleichheitstest mit beidseitigem Fehler $0 < \varepsilon < \frac{1}{2}$ realisiert. Betrachte den Algorithmus **ALG'** für t ungerade:

- Lasse **ALG** t Mal laufen und notiere die Ergebnisse als 0 (für Ausgabe $\gg A \not\equiv B \ll$) und 1 (für Ausgabe $\gg A \equiv B \ll$).
- Kommt im Ergebnisvektor 0 öfter vor als 1, so gebe 0 aus, ansonsten 1.

(Majoritätsvotum)

Wir wollen die (garantierte) Fehlerwahrscheinlichkeit ε' von **ALG'** abschätzen: O.B.d.A. sei $A \equiv B$, d.h. mit Wahrscheinlichkeit $\delta_+ \geq 1 - \varepsilon$ gibt **ALG** eine 1 aus und mit Wahrscheinlichkeit $\delta_- \leq \varepsilon$ eine 0 ($\delta_+ + \delta_- = 1$). Ausgabe von **ALG'** sei 0, d.h. im Ergebnisvektor kommt maximal $\lfloor \frac{t}{2} \rfloor$ Mal eine 1 vor. Dann gilt:

$$\begin{aligned}
\varepsilon' &\leq \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \binom{t}{i} \delta_+^i \delta_-^{t-i} \\
&\leq \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \binom{t}{i} \delta_+^i \delta_-^{t-i} \underbrace{\left(\frac{\delta_+}{\delta_-} \right)^{\frac{t}{2}-i}}_{\geq 1, \varepsilon \leq \frac{1}{2} \wedge i \leq \frac{t}{2}} \\
&= (\delta_+ \delta_-)^{\frac{t}{2}} \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \binom{t}{i} \\
&\leq (\delta_+ \delta_-)^{\frac{t}{2}} \sum_{i=0}^t \binom{t}{i} \\
&= \left(2\sqrt{\delta_+ \delta_-} \right)^t \\
&\leq \underbrace{\left(2\sqrt{\varepsilon(1-\varepsilon)} \right)^t}_{< 1, \varepsilon < \frac{1}{2}} \\
&\leq (4\varepsilon)^{\frac{t}{2}} \quad \text{für } \varepsilon < \frac{1}{4}
\end{aligned}$$

Kapitel 2

Grundlegende Datenstrukturen

2.1 Stacks und Queues

Stack: ist eine Datenstruktur, bei der Einfügen und Entfernen dem LIFO – Prinzip folgen.
Operationen auf Stack S :

- **PUSH**(o) fügt das Objekt o an der Spitze von S ein.
- **POP** entfernt das oberste Objekt von S und gibt es zurück.
- **TOP** gibt das oberste Objekt von S zurück, ohne es zu entfernen.

Implementierung eines Stacks: z.B. Feld (beachte Überläufe); **POP**, **TOP** in $O(1)$ amortisiert, **PUSH** in $O(1)$ amortisiert bei multiplikativer Vergrößerungsstrategie.

Queue: ist eine Datenstruktur, bei der Einfügen und Entfernen dem FIFO – Prinzip folgen.
Operationen auf Queue Q :

- **ENQUEUE**(o) fügt das Objekt o am Ende von Q ein.
- **DEQUEUE** entfernt das erste Objekt von Q und gibt es zurück.
- **FRONT** gibt das erste Objekt von Q zurück, ohne es zu löschen.

Implementierung einer Queue: z.B. Feld; **dequeue**, **front** in $O(1)$, **enqueue** in $O(1)$ amortisiert.

2.2 Bäume

DS1: Ein Baum ist ein ungerichteter, zusammenhängender Graph. Hier: Ein (gewurzelter) Baum T ist eine Menge von Knoten, die in einer Eltern – Kind – Beziehung mit folgenden Eigenschaften stehen:

- T hat einen ausgezeichneten Knoten r (**WURZEL** von T).
- Jeder Knoten $v \neq r$ hat einen **ELTERNKNOTEN** u .

Bezeichnungen:

- Ist u Elternknoten von v , so ist v **KIND** von u .
- Knoten ohne Kinder heißen **BLÄTTER**.
- Knoten mit mindestens einem Kind heißen **INNERE KNOTEN**.
- Ein **NACHFAHRE** von v ist entweder v selbst oder ein Nachfahre eines Kindes von v .
- **VORFAHREN** von v sind alle Knoten, für die v ein Nachfahre ist.

- Ein TEILBAUM von T mit Wurzel von v besteht aus allen Nachfahren von v in T .
- Ein Baum T ist GEORNET, falls für alle Knoten alle Kinder total geordnet werden können.
- Ein (ECHTER) BINÄRER BAUM T ist ein geordneter Baum, bei dem alle inneren Knoten höchstens (genau) zwei Kinder haben.

Operationen auf Bäumen T :

- PARENT(v) gibt den Elternknoten von v zurück (Fehlermeldung, falls v die Wurzel ist).
- CHILD(v) gibt einen »Iterator« der Kinder von v zurück.
- ROOT gibt die Wurzel von T zurück.

Ein Iterator besteht aus einer Folge, einer aktuellen Position (Index) in dieser Folge und einer Methode `next` zur Bestimmung der nächsten Position und Aktualisierung der gegenwärtigen Position.

Traversierung

TRAVERSIERUNG ist das »Ablaufen« aller Knoten in einem Baum (und Ausführen von Operationen auf den Knoten).

Beispiel Bestimmung der Höhe eines Baumes

- Die TIEFE d_v eines Knotens v ist die Anzahl seiner Vorfahren (ohne v).
- Die HÖHE $h(T)$ eines Baumes T ist die maximale Tiefe eines Knotens in T .

Erster Ansatz: Bestimme die Tiefe aller Knoten und nehme das Maximum: $O(n^2)$.

Zweiter Ansatz: Verwende rekursive Charakterisierung von $h(T)$:

1. Ist v ein Blatt, so ist $h(T_v) = 0$.
2. Ist v ein innerer Knoten, dann ist

$$h(T_v) = 1 + \max_{w \text{ Kind von } v} h(T_w)$$

3. $h(T) = h(T_r)$

Komplexität: (sei c_v die Anzahl der Kinder von v)

$$O\left(\sum_{v \in T} 1 + c_v\right) = O\left(n + \underbrace{\sum_{v \in T} c_v}_{n-1}\right) = O(n)$$

Eulertour

(DS1: Eine EULERTOUR in G ist ein Weg, der jede Kante von G genau einmal enthält und bei der Start- und Endknoten identisch sind. Ein zusammenhängender Graph G hat eine Eulertour genau dann, wenn alle Knotengrade gerade sind.)

Hier:

- Betrachte jede Kante als Doppelkante (damit haben alle Knoten geraden Grad).
- Wir interessieren uns für eine Eulertour von der Wurzel zur Wurzel.

Algorithmus: `Eulertour(T, v)`

```

1 Werte v aus (bevor linkes Kind besucht wird)
2 IF v innerer Knoten
3     Eulertour( T, linkes Kind von v )
4 Werte v aus (bevor rechtes Kind besucht wird)
5 IF v innerer Knoten
6     Eulertour( T, rechtes Kind von v )
7 Werte v aus (nach Rueckkehr vom rechten Kind)

```

Laufzeit dieses Algorithmus: $O(n)$.

Spezialfälle:

- preorder: ohne Zeilen 4 und 7
- postorder: ohne Zeilen 1 und 4
- inorder: ohne Zeilen 1 und 7

Implementierung von binären Bäumen

1. Arrays:

Definiere Nummerierung $p : T \rightarrow \mathbb{N}$

- (a) v Wurzel $\Rightarrow p(v) = 1$
- (b) v linkes Kind von $u \Rightarrow p(v) = 2p(u)$
- (c) v rechtes Kind von $u \Rightarrow p(v) = 1 + 2p(u)$

(p ist Level – Nummerierung)

Speichere Inhalt von Knoten v an Position $p(v)$ in einem Array; Größe N ist $\max_{v \in T} p(v)$.

Es gilt:

- p ist injektiv
- $N \leq 2^{\frac{n+1}{2}} - 1$ für echten binäre Baum (\Rightarrow im schlechtesten Fall exponentieller ungenutzter Speicher)

Operationen:

- (a) **root**, **parent**, **child** in $O(1)$ im uniformen Kostenmodell.
- (b) **parent**, **child** in $O(|p(v)|)$ bzw. $O(n)$, wenn n die Anzahl der Knoten in T ist, im logarithmischen Kostenmodell.

2. Linkstruktur:

Knoten v als Struktur mit Zeigern auf **parent**(v), linkes und rechtes Kind von v und den Inhalt (für allgemeine Bäume: doppelt verkettete Liste mit Zeigern auf die Kinder).

Operationen:

- (a) **root**, **parent** in $O(1)$, **child** in $O(c_v)$.

2.3 Priority Queues und Heaps

Wir identifizieren ein abzuspeicherndes Objekt mit einem Schlüssel aus einem Universum U (z.B. $U = \mathbb{N}$). Dabei setzen wir voraus, dass die Schlüssel total geordnet werden können, d.h.

- Reflexivität: $k \leq k$
- Antisymmetrie: $k_1 \leq k_2 \wedge k_2 \leq k_1 \Rightarrow k_1 = k_2$
- Transitivität: $k_1 \leq k_2 \wedge k_2 \leq k_3 \Rightarrow k_1 \leq k_3$
- Totalität: $k_1 \leq k_2 \vee k_2 \leq k_1$

⇒ Jede endliche Teilmenge von U hat einen kleinsten Schlüssel k_{\min} . Eine PRIORITY QUEUE P ist eine Datenstruktur, die Objekte mit Schlüsseln verbindet und folgende Operationen unterstützt:

- `insert(o, k)` fügt das Objekt o mit Schlüssel k in P ein.
- `extractMin` entfernt das Objekt mit kleinstem Schlüssel aus P und gibt es zurück.
- `minObject` gibt das Objekt, das zum kleinsten Schlüssel gehört, zurück, ohne es zu entfernen.
- `minKey` gibt den kleinsten Schlüssel zurück.

Bemerkung: Priority Queues sind inhaltsbezogen — im Gegensatz zu Stacks, Queues und Bäumen.

Sortieren mit Priority Queues

Gegeben: Array $A[1..n]$ von Objekten, die total geordnet werden können.

Gesucht: Aufsteigend sortiertes Feld $A[1..n]$

Algorithmus: PQ-Sort(A, P)

```

1  FOR i := 1 TO n
2    P.insert( A[i], A[i] )
3  FOR i := 1 TO n
4    A[i] := P.extractMin
```

Die Komplexität von PQ-Sort hängt ab von:

- der amortisierten Komplexität t_n von `insert` ⇒ Zeilen 1 und 2 in $O(n \cdot t_n)$
- und der amortisierten Komplexität t'_n von `extractMin` ⇒ Zeilen 3 und 4 in $O(n \cdot t'_n)$

Einfache Implementierungen von Priority Queues:

1. ungeordnetes Array: Einfügen am Ende des Feldes, Extrahieren durch Suche und Elimination leerer Feldeinträge.
`insert`: $O(1)$ amortisiert
`extractMin`: $\Theta(n)$ amortisiert
⇒ PQ-Sort (Selectionsort): $O(n^2)$
2. geordnetes Array: Einfügen an richtige Position i im Feld (links von i nur kleinere Schlüssel, rechts nur größere), Extrahieren vom Anfang her
`insert`: $\Theta(n)$
`extractMin`: $O(1)$ amortisiert
⇒ PQ-Sort (Insertionsort): $O(n^2)$

Priority Queue – Implementierung mit Heaps

Ein Heap ist ein echter Binärbaum, bei dem in den inneren Knoten Schlüssel gespeichert werden können und der zusätzlich zwei Eigenschaften erfüllt:

1. relationale Eigenschaft zwischen den Schlüsseln in T : »Heap – Ordnung« (Heap – Eigenschaft): In einem Heap T gilt für alle inneren Knoten v : v nicht Wurzel ⇒ $\text{key}(v)$ ist größer als $\text{key}(\text{parent}(v))$.
2. strukturelle Eigenschaft des Baumes: »vollständiger Baum«: Ein Binärbaum T der Höhe h heißt VOLLSTÄNDIG, falls alle Level $0, 1, 2, \dots, h-1$ die maximal mögliche Anzahl von Knoten (d.h. 2^i für Level i) besitzen und sich in Level $h-1$ alle inneren Knoten links von Blättern befinden.

Damit enthält die Wurzel eines Heaps immer das Minimum.

Satz 3 Für einen Heap T über n Objekten gilt $h(T) = \lceil \log(n+1) \rceil$.

Beweis. Es gilt:

1.

$$n \geq \left(\sum_{i=0}^{h(T)-2} 2^i \right) + 1 = 2^{h(T)-1} - 1 + 1 = 2^{h(T)-1}$$

2.

$$n \leq \sum_{i=0}^{h(T)-1} 2^i = 2^{h(T)} - 1$$

Aus 1 folgt: $h(T) \leq 1 + \log n < 1 + \log(n+1)$. Aus 2 folgt $h(T) \geq \log(n+1)$. Damit gilt $h(T) = \lceil \log(n+1) \rceil$. \square

Ziel: Realisiere `insert` und `extractMin` in $O(h(t))$.

1. Einfügen; d.h. aus Heap T ohne Schlüssel k und Schlüssel k produziere Heap T' mit Schlüssel k .

Algorithmus: `Insert(T, k)`

```

1  Bestimme Blatt v mit p( v ) minimal
2  Haenge 2 Kinder an v an
3  Setze Inhalt von v auf k
4  WHILE ( v nicht Wurzel und Inhalt von v < Inhalt von parent( v ) )
5    Vertausche Inhalt von v und parent( v )
6    v := parent( v )
```

\Rightarrow Komplexität: $O(h(t))$ im worst - case

2. `extractMin`, d.h. entferne Minimum aus Wurzel des Heaps T und produziere Heap T' .

Algorithmus: `ExtractMin(T)`

```

1  s := Inhalt von Wurzel r
2  Bestimme inneren Knoten v* mit p( v* ) maximal
3  Setze Inhalt von Wurzel r auf Inhalt von v*
4  Entferne beide Kinder von v*
5  v := r
6  WHILE ( v innerer Knoten und
7    Inhalt von v > min{ Inhalt der Kinder von v } )
8    Vertausche Inhalt von v mit Inhalt des minimalen Kindes von v
9    v := minimales Kind von v
```

\Rightarrow Komplexität: $O(h(t))$ im worst - case

Damit gilt: `insert` und `extractMin` in $O(\log n)$. PQ-Sort mit Heaps (Heapsort) in $O(n \log n)$

Bottom - Up Heap - Konstruktion

Iteratives Einfügen von n Elementen in einen anfänglich leeren Heap kostet $O(n \log n)$ im worst - case.

Bemerkung: Das gilt auch amortisiert, denn:

$$\sum_{i=1}^n \log i = \log(n!) = \Theta(n \log n)$$

Bottom - Up - Ansatz (rekursiv) liefert den Heap in $O(n)$. Zur Vereinfachung sei $n = 2^h - 1$, damit der Heap vollständig in allen Leveln ist, $h = \log(n+1)$.

Algorithmus: `BottomUpHeap(S)`: Eingabe: Folge S mit $n = 2^h - 1$ Schlüsseln, Ausgabe: Heap T mit den Schlüsseln von S .

```

1  IF S leer
2    RETURN leerer Heap
3  Entferne ersten Schluessel k aus S
4  Spalte S in Folgen S1, S2 auf mit jeweils (n-1)/2 Elementen
5  T1 := BottomUpHeap( S1 )
6  T2 := BottomUpHeap( S2 )
7  Bilde Baum T mit Wurzel v, die k enthaelt, und den Teilbaeumen T1 und T2
8  Vertausche Inhalt von v mit minimalem Inhalt der Kinder von v (iterativ)
9  RETURN T

```

Es gilt: `BottomUpHeap` benötigt für n Schlüssel Laufzeit $O(n)$. Begründung: Es sei T der resultierende Heap, v ein innerer Knoten und T_v der Teilbaum mit Wurzel v . Im worst – case muss der Inhalt von Wurzel v bis in den niedrigsten Knoten verschoben werden, d.h. die Formierung von T_v aus den Teilbäumen der Kinder ist $O(h(T_v))$. Sei nun $P(v)$ ein Pfad in T von v bis zum nächsten Knoten in Inorder. $P(v)$ repräsentiert das Absenken des Inhaltes von v bis zum niedrigsten Knoten. Damit: Komplexität:

$$\begin{aligned}
& O\left(\sum_{v \in T} \text{Länge von } P(v)\right) \\
& = O(\text{Anzahl der Kanten in } T) \\
& = O(n)
\end{aligned}$$

2.4 Wörterbuch und Hashing

Gegeben seien Objekte o_1, \dots, o_m , die durch zugehörige Schlüssel k_1, \dots, k_m identifiziert werden können. Unisversum $U = \{0, 1, \dots, N-1\}$ (natürliche Zahlen zur Vereinfachung), aus dem Schlüssel kommen können; $m \ll N = \|U\|$. Ein WÖRTERBUCH (dictionary) ist eine Datenstruktur, die folgende Operationen unterstützt:

1. `Find(k)` testet, ob für k ein Objekt im Wörterbuch enthalten ist; wenn ja, gibt es das Objekt zurück, wenn nein das leere Objekt (NIL).
2. `Insert(o, k)` fügt das Objekt o mit Schlüssel k in das Wörterbuch ein (unter Voraussetzung, dass kein anderes Objekt mit Schlüssel k enthalten ist).
3. `Delete(k)` entfernt ein Objekt mit Schlüssel k aus dem Wörterbuch und gibt es als Ergebnis zurück.

Die Kapazität (Größe) eines Wörterbuchs wird mit n bezeichnet (typischerweise $m \ll n \ll N$).

Idee des Hashing: Implementiere das Wörterbuch als (unsortiertes) Array der Größe $n = O(m)$ (meist $m \leq n \leq 2m$) und lasse die Position eines Datensatzes vom Wert des Schlüssels abhängen.

Eine Funktion $h : \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, n-1\}$ heißt HASHFUNKTION. Für $k \in U$ ist dann $h(k)$ die Position des Datensatzes mit Schlüssel k im Wörterbuch.

Problem: Da $n \ll N$, kann die Hashfunktion nicht injektiv sein, d.h. es gibt $k_1, k_2 \in U$ mit $k_1 \neq k_2$ und $h(k_1) = h(k_2)$. Werden zwei Schlüssel auf die gleiche Position abgebildet, so sprechen wir von einer KOLLISSION.

→ Kollisionen sind nicht selten!

Satz 4 In einer Hashtabelle der Größe n mit m Einträgen sei für jeden Schlüssel jede Hashposition gleich wahrscheinlich. Dann gibt es mit Wahrscheinlichkeit

$$\geq 1 - e^{-\frac{m(m-1)}{2n}} \left(\approx 1 - e^{-\frac{m^2}{2n}} \right)$$

Beweis. Es sei A_m das Ereignis, dass unter m Schlüsseln keine Kollision auftritt.

$$\begin{aligned} \mathcal{P}(A_m) &= \prod_{j=0}^{m-1} \frac{n-j}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right) \\ &\leq \prod_{j=0}^{m-1} e^{-\frac{j}{n}} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \end{aligned}$$

Es folgt die Behauptung. □

Korollar 5 Sind in einer Hashtabelle der Größe n mindestens $\omega(\sqrt{n})$ Einträge enthalten und ist für jeden Schlüssel jede Hashposition gleich wahrscheinlich, so tritt mit Wahrscheinlichkeit $1 - o(1)$ eine Kollision auf.

Bemerkungen

1. Um die Kollisionszahl gering zu halten, müssen Hashfunktionen gut streuen.
2. Typische Hashfunktionen:

$$\begin{aligned} h(k) &= k \pmod n && \text{(Teilermethode)} \\ h(k) &= \lfloor n(ak - \lfloor ak \rfloor) \rfloor \text{ für } a < 1 && \text{(Multiplikationsmethode)} \end{aligned}$$

2.4.1 Strategien zur Kollisionsauflösung

Geschlossene Hashverfahren

Der Schlüssel wird unter der durch die Hashfunktion bestimmten Stelle abgespeichert bzw. gefunden.

→ Hashing durch Verkettung (chaining)

Idee: Kollisionen werden nicht aufgelöst, d.h. hinter jeder Hashposition steht eine verkettete Liste, ein neues Element wird immer an den Listenanfang gesetzt.

Im schlechtesten Fall werden alle m Schlüssel auf die gleiche Position abgebildet. Deshalb für allgemeine Strategie A zur Kollisionsauflösung:

$A^+ =_{\text{def}}$ mittlerer Zeitbedarf für erfolgreiche Suche unter Verwendung von A .

$A^- =_{\text{def}}$ mittlerer Zeitbedarf für erfolglose Suche unter Verwendung von A .

Ein Zugriff auf die Hashtabelle heißt SONDIERUNG. Der FÜLLFAKTOR α einer Hashtabelle ist definiert als $\alpha =_{\text{def}} \frac{m}{n}$.

- A^- für Chaining: mittlere Länge der n Listen: $\frac{m}{n} = \alpha \Rightarrow A^- \leq 1 + \alpha$.
- A^+ für Chaining:

$$\begin{aligned} A^+ &= \frac{1}{m} \sum_{i=0}^{m-1} \left(1 + \frac{i-1}{n} + \frac{1}{n}\right) \\ &= \frac{1}{m} \sum_{i=0}^{m-1} \left(1 + \frac{i}{n}\right) \\ &= 1 + \frac{m(m-1)}{2nm} \\ &\leq 1 + \frac{m}{2n} = 1 + \frac{\alpha}{2} \end{aligned}$$

Für festen Füllfaktor α ergibt sich also im Mittel $\Theta(1)$ Laufzeit.

Offene Hashverfahren

Idee: Bei Kollision wird so lange weitergesucht, bis eine freie Hash - Position (beim Einfügen) gefunden ist.

Beispiel

- Linear Probing (lineares Sondieren)

$$\bar{h}(k, i) =_{\text{def}} (h(k) + i) \pmod n$$

- Double Hashing

$$\bar{h}(k, i) =_{\text{def}} (h(k) + ih'(k)) \pmod n$$

wobei $h'(k)$ für alle k teilerfremd zu n (n Primzahl).

Wir analysieren offene Hashverfahren unter der Annahme des gleichverteilten Hashings, d.h. Folgen $(\bar{h}(k, 0), \bar{h}(k, 1), \dots, \bar{h}(k, n-1))$ sind gleich wahrscheinliche Permutationen von $(0, 1, \dots, n-1)$.

- A^- für Open Hashing: Sei x die Anzahl der Sondierungen für erfolglose Suche mit A . Definiere A_i als Ereignis, dass die i -te Sondierung zu einem belegten Feldelement führt. Damit:

$$\begin{aligned} \mathbb{P}(x \geq i) &= \mathbb{P}(A_1 \cap A_2 \cap \dots \cap A_{i-1}) \\ &= \mathbb{P}(A_1) \cdot \mathbb{P}(A_2|A_1) \cdot \dots \cdot \mathbb{P}(A_{i-1}|A_1 \cap \dots \cap A_{i-2}) \\ &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \dots \cdot \frac{m-i+2}{n-i+2} \\ &\stackrel{m \leq n}{\leq} \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} \end{aligned}$$

$$\begin{aligned} A^- &= \mathbb{E}x \leq \sum_{i=1}^{\infty} i\mathbb{P}[x = i] = \sum_{i=1}^{\infty} \mathbb{P}(x \geq i) \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} \end{aligned}$$

- A^+ für Open Hashing: Erfolgreiche Suche nach Schlüssel k im $(i+1)$ -ten Schritt bedeutet erfolglose Suche bis zum i -ten Schritt (Schlüssel k wurde als $(i+1)$ -ter Schlüssel eingefügt); d.h. im Mittel

$$\frac{1}{1 - \frac{i}{n}} = \frac{n}{n-i} \quad (\text{nach Beweis von } A^-)$$

Damit:

$$\begin{aligned} A^+ &= \frac{1}{m} \sum_{i=0}^{m-1} \frac{n}{n-i} = \frac{n}{m} \sum_{i=0}^{m-1} \frac{1}{n-i} \\ &= \frac{n}{m} \left(\sum_{i=n-m+1}^n \frac{1}{i} \right) \\ &\leq \frac{n}{m} \int_{n-m}^n \frac{1}{x} dx \\ &= \frac{n}{m} (\ln n - \ln(n-m)) = \frac{n}{m} \ln \frac{n}{n-m} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

Beispiel

- Lineares Sondieren:

$$\begin{aligned} A^+ &\approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \\ A^- &\approx \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right) \end{aligned}$$

- Double Hashing:

$$A^- \approx \frac{1}{1 - \alpha}$$

$$A^+ \approx \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

Universelles Hashing

Idee: Zufällige Wahl einer Hashfunktion zur Laufzeit aus einer Menge von Hashfunktionen (Carter, Wegmann, 1979)

Eine Familie H von Hashfunktionen heißt UNIVERSSELL, wenn für alle $x, y \in U$ mit $x \neq y$ gilt:

$$\frac{|\{h \in H : h(x) = h(y)\}|}{\|H\|} \leq \frac{1}{n}$$

wobei $h : U \rightarrow \{0, 1, \dots, n-1\}$ für alle $h \in H$ gilt.

Satz 6 Es sei H eine universelle Familie von Hashfunktionen (für eine Hashtabelle der Größe n) und es sei $h \in H$ eine unter Gleichverteilung zufällig gewählte Hashfunktion aus H . Für eine Menge $M \subseteq U$ von $m \leq n$ Schlüsseln ist die erwartete Anzahl von Kollisionen eines festen Schlüssels $x \in M$ mit einem anderen Element aus M kleiner als 1.

Beweis. Es sei $C_x(y)$ definiert als

$$C_x(y) =_{\text{def}} \begin{cases} 1 & \text{falls } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$$

Da H universell und $h \in H$ zufällig unter Gleichverteilung gewählt ist, gilt

$$\begin{aligned} \mathbb{E}C_x(y) &= 0\mathbb{P}[h(x) \neq h(y)] + 1\mathbb{P}[h(x) = h(y)] \\ &= \mathbb{P}[h(x) = h(y)] \leq \frac{1}{n} \end{aligned}$$

Für $C_x =_{\text{def}} \sum_{\substack{y \in M \\ x \neq y}} C_x(y)$ gilt somit

$$\mathbb{E}C_x = \sum_{\substack{y \in M \\ x \neq y}} C_x(y) \leq \frac{m-1}{n} \leq 1$$

□

Gibt es universelle Familien von Hashfunktionen? Es sei $U = \{0, 1, \dots, n+1\}^{r+1}$ mit n Primzahl. Definiere

$$H =_{\text{def}} \{h_\alpha : h_\alpha : U \rightarrow \{0, 1, \dots, n-1\}, \alpha \in U\}$$

wobei

$$h_\alpha(x_0, x_1, \dots, x_r) =_{\text{def}} \left(\sum_{i=0}^r \alpha_i x_i \right) \bmod n$$

Es gilt: H ist universell.

Begründung: Es seien $x, y \in U$ mit $x \neq y$, d.h. o.B.d.A. $x_0 \neq y_0$. Ist $h_\alpha(x) = h_\alpha(y)$ für $\alpha \in U$, so gilt

$$\alpha_0(y_0 - x_0) \equiv \sum_{i=1}^r \alpha_i(x_i - y_i) \pmod{n}$$

Da n Primzahl (d.h. $(\{0, 1, \dots, n-1\}, +, \cdot)$ Körper) und $y_0 - x_0$ bzw. $\sum_{i=1}^r \alpha_i(x_i - y_i)$ fest, gibt es genau ein α_0 , so dass die obige Gleichung erfüllt ist. Sind also $\alpha_1, \dots, \alpha_r$ beliebig, so gibt es genau ein α , so dass $h_\alpha(x) = h_\alpha(y)$ gilt. Da x, y fest gewählt sind, gibt es genau n^r Möglichkeiten, α zu wählen mit $h_\alpha(x) = h_\alpha(y)$. Damit:

$$\frac{|\{h \in H : h(x) = h(y)\}|}{\|H\|} = \frac{n^r}{n^{r+1}} = \frac{1}{n}$$

Beispiel $n = 2, r = 2, U = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

$$h_{110}(x_0, x_1, x_2) = x_0 \oplus x_1$$

Seien $x = 010, y = 110$. Bestimme α mit $h_\alpha(010) = h_\alpha(110)$. $\alpha_1 \equiv \alpha_0 + \alpha_1 \pmod{2} \Rightarrow \alpha_0 \equiv 0 \pmod{2}$, d.h. $\{h_\alpha : h_\alpha(x) = h_\alpha(y)\} = \{h_{0\alpha_1\alpha_2} : \alpha_1, \alpha_2 \in \{0, 1\}\}$

Bemerkung: Für $U = \{0, 1, \dots, N-1\}$ und Primzahl $n \geq 2$ interpretiere Schlüssel k als n -äre Darstellung (k_0, k_1, \dots, k_r) , d.h. $k = \sum_{i=0}^r k_i n^i$ (wobei $N < n^{r+1}$ sein sollte).

Wie groß müssen die Familien H sein?

- H aus dem Beispiel: $\|H\| = n^{r+1} = \|U\|$
- Es gibt weitere Klassen der Größen $n^{\log \|U\|}, \|U\|^{\log n}$

Satz 7 Es sei H eine universelle Familie von Hashfunktionen mit $h : U \rightarrow \{0, 1, \dots, n-1\}$. Dann gilt

$$\|H\| \geq n \left\lfloor \frac{\log \|U\| - 1}{\log n} \right\rfloor$$

Beweis. Es sei $H = \{h_1, h_2, \dots, h_t\}$. Betrachte eine Folge $U = U_0 \supseteq U_1 \supseteq U_2 \supseteq \dots \supseteq U_t$, die wie folgt definiert ist:

$$U_i \stackrel{\text{def}}{=} U_{i-1} \cap h_i^{-1}(y_i)$$

mit $y_i \in \{0, 1, \dots, n-1\}$ so gewählt, dass $\|U_i\|$ maximal ist. Damit:

- h_i ist auf U_i konstant
- $\|U_i\| \geq \frac{\|U_{i-1}\|}{n}$ (d.h. $\|U_i\| \geq \frac{\|U\|}{n^i}$)

Sei nun $t_0 = \left\lfloor \frac{\log \|U\| - 1}{\log n} \right\rfloor$. Dann gilt: $\|U_{t_0}\| \geq 2$. Begründung:

$$\log \|U_{t_0}\| \geq \log \|U\| - t_0 \log n \geq \log \|U\| - \left(\frac{\log \|U\| - 1}{\log n} \right) \log n = 1$$

Es seien $x, y \in U_{t_0}$ mit $x \neq y$. Dann gilt:

$$t_0 \leq \|\{h \in H : h(x) = h(y)\}\| \leq \frac{\|H\|}{n}$$

Damit:

$$\|H\| \geq nt_0 = n \left\lfloor \frac{\log \|U\| - 1}{\log n} \right\rfloor$$

□

Kapitel 3

Suchbäume und Skiplisten

Geordnetes Wörterbuch: Wörterbuch mit totaler Ordnung auf den Schlüsseln
unterstützte Operationen:

- `find(k)`, `insert(o, k)`, `delete(k)`
- `ClosestKey≤(k)`, `ClosestKey≥(k)`: gibt den größten (kleinsten) Schlüssel $\leq k$ ($\geq k$) zurück, der sich im Wörterbuch befindet

Implementierung von geordneten Wörterbüchern

1. Tabellen, geordnet nach Schlüsseln:

- `find`: $O(\log n)$ mittels binärer Suche
- `insert`, `delete`: $O(n)$
- `ClosestKey`: $O(\log n)$

2. Suchbäume:

- interne Suchbäume: Schlüssel werden in inneren Knoten abgespeichert
- externe Suchbäume: Schlüssel (Objekte) werden in Blättern abgespeichert

3.1 Binäre Suchbäume

Ein BINÄRER SUCHBAUM T ist ein echter Binärbaum, in dem jeder innere Knoten v ein Paar (o, k) des Wörterbuchs enthält und für alle Schlüssel k' im linken Teilbaum von v $k' \leq k$ sowie für alle Schlüssel k' im rechten Teilbaum von v $k' \geq k$ gilt.

Annahme: Alle Schlüssel sind verschieden.

Damit lässt sich folgender Suchalgorithmus ausführen:

Algorithmus: `Search(k, v)`

- Eingabe: Schlüssel k und Knoten v eines binären Suchbaumes T
- Ausgabe: Knoten w des Teilbaumes T_v , so dass entweder w innerer Knoten mit Schlüssel k ist oder w ein Blatt ist, das den Schlüssel k enthalten müsste, falls k im Baum enthalten wäre.

```
1 IF v ist Blatt
2   RETURN v
3 IF k = key( v )
4   RETURN v
5 ELSE
6   IF k < key( v )
```

```

7     RETURN Search( k, linkes Kind von v )
8     ELSE
9     RETURN Search( k, rechts Kind von v )

```

Komplexität von **Search**: $O(h(T))$ im worst case \Rightarrow **find**, **ClosestKey** in $O(h(T))$ im worst case. Klar: **insert** (mit Finden der Position) in $O(h(T_v))$ im worst case. Delete (mit Finden des Knotens mit Schlüssel k):

1. **Search** gibt Blatt zurück \Rightarrow Schlüssel k kommt in T nicht vor.
2. **Search** gibt einen Knoten w mit einem Blatt als Kind zurück \Rightarrow entferne Knoten w mit Kind z und hänge anderes Kind von w an an **parent**(w) (falls w Wurzel ist, dann sind wir bereits ohne Einhängen fertig).
3. **Search** gibt einen inneren Knoten v zurück, dessen Kinder ebenfalls innere Knoten sind \Rightarrow entferne Knoten mit nächstgrößerem Schlüssel als **key**(v) und setze Inhalt von v auf k' ; hänge rechten Teilbaum von v an **parent**(w) ein.
Komplexität: $O(h(T))$ für **Search**, im schlechtesten Fall (z.B. füge n Schlüssel aufsteigend sortiert in T ein): **find**, **insert**, **delete** in $O(n)$.

Average – case – Analyse: »Wie hoch ist ein Suchbaum im Mittel?« (d.h. Abschätzung von $\mathbb{E}(\max_{v \in T} d_v)$)

$$\begin{aligned}
 \mathbb{E} \left(\max_{v \in T} d_v \right) &\leq \log \mathbb{E} \left(2^{\max_{v \in T} d_v} \right) \\
 &= \log \mathbb{E} \left(\max_{v \in T} 2^{d_v} \right) \\
 &\leq \log \mathbb{E} \left(\sum_{v \in T} 2^{d_v} \right)
 \end{aligned}$$

Es gilt:

$$\begin{aligned}
 \mathbb{E} \left(\sum_{v \in T} 2^{d_v} \right) &= 2^0 + \frac{1}{\|T\|} \left(\sum_{v \in T} 2 \mathbb{E} \left(\sum_{u \in T_{\text{linkes Kind von } v}} 2^{d_u} \right) + 2 \mathbb{E} \left(\sum_{u \in T_{\text{rechtes Kind von } v}} 2^{d_u} \right) \right) \\
 &= 1 + \frac{1}{\|T\|} \sum_{v \in T} 2 \left(\mathbb{E} \left(\sum_{u \in T_{\text{linkes Kind von } v}} 2^{d_u} \right) + \mathbb{E} \left(\sum_{u \in T_{\text{rechtes Kind von } v}} 2^{d_u} \right) \right)
 \end{aligned}$$

Definiere

$$g(\|T\|) =_{\text{def}} \mathbb{E} \left(\sum_{v \in T} 2^{d_v} \right)$$

Dann gilt:

$$g(\|T\|) = 1 + \frac{1}{\|T\|} \sum_{v \in T} (g(\|T_{\text{linkes Kind von } v}\|) + g(\|T_{\text{rechtes Kind von } v}\|))$$

oder in n ausgedrückt:

$$\begin{aligned}
 g(n) &= 1 + \frac{1}{n} \sum_{i=1}^n 2(g(i-1) + g(n-i)) \\
 &= 1 + \frac{4}{n} \sum_{i=0}^{n-1} g(i) \\
 &\Rightarrow ng(n) = n + 4 \sum_{i=0}^{n-1} g(i) \tag{3.1}
 \end{aligned}$$

Aus 3.1 folgt weiter

$$(n-1)g(n-1) = n-1 + 4 \sum_{i=0}^{n-2} g(i) \quad (3.2)$$

Damit (3.1 - 3.2):

$$\begin{aligned} ng(n) - (n-1)g(n-1) &= n - (n-1) + 4g(n-1) \\ \Rightarrow ng(n) &= 1 + (n+3)g(n-1) \\ \Rightarrow g(n) &= \frac{n+3}{n}g(n-1) + \frac{1}{n} \\ &= \frac{n+3}{n} \frac{n+2}{n-1}g(n-2) + \frac{1}{n} + \frac{n+3}{n} \frac{1}{n-1} \\ &= \left(\prod_{i=0}^{n-2} \frac{n+3-i}{n-i} \right) g(1) + \sum_{j=0}^{n-1} \frac{1}{n} \prod_{i=1}^j \frac{n+4-i}{n-i} \\ &\leq \frac{(n+3)(n+2)(n+1)}{4 \cdot 3 \cdot 2} + \sum_{j=0}^{n-1} \frac{1}{n} \frac{(n+3)(n+2)(n+1)n}{4 \cdot 3 \cdot 2} \\ &= (n+1) \frac{(n+3)(n+2)(n+1)}{4 \cdot 3 \cdot 2} \leq (n+1)n^3 \leq 2n^4 \end{aligned}$$

Damit gilt:

$$\begin{aligned} \mathbb{E} \left(\max_{v \in T} d_v \right) &\leq \log \mathbb{E} \left(\sum_{v \in T} 2^{d_v} \right) = \log g(\|T\|) \\ &\leq \log 2n^4 = 4 \log n + 4 = O(\log n) \end{aligned}$$

Im Mittel hat jeder binäre Suchbaum also nur logarithmische Höhe. `find`, `ClosestKey`, `insert`, `delete` laufen im average case in $O(\log n)$.

3.2 AVL – Bäume

(1962 von Adelson-Velskii und Landis eingeführt)

Ziel: Begrenze die Höhe von binären Suchbäumen.

Es sei T ein binärer Suchbaum. Ein Knoten von T heißt HÖHENBALANCIERT genau dann, wenn sich die Höhen der an seinen Kindern hängenden Teilbäume um höchstens 1 unterscheiden. T ist ein AVL – BAUM genau dann, wenn jeder Knoten höhenbalanciert ist.

Satz 8 Ein AVL – Baum der Höhe h hat mindestens

$$\left(\frac{1 + \sqrt{5}}{2} \right)^h (\approx 1.618^h)$$

Knoten.

Beweis. (Induktion über h)

(IA) $h = 0$: Einziger AVL – Baum der Höhe 0 ist der Baum mit genau einem Knoten. Damit:

$$1 = \left(\frac{1 + \sqrt{5}}{2} \right)^0$$

$h = 1$: Ein AVL – Baum der Höhe 1 besitzt zwei Kanten, d.h. drei Knoten. Damit:

$$\left(\frac{1 + \sqrt{5}}{2} \right)^1 \leq 3$$

(IS) Es sei $h \geq 2$. Seien T ein (beliebiger) AVL – Baum der Höhe h , x die Wurzel von T und T_l und T_r der linke bzw. rechte Teilbaum der Wurzel. Da T als AVL – Baum ein binärer Suchbaum ist, existieren wegen $h \geq 1$ T_l und T_r stets. Die Höhen von T_l und T_r sind mindestens $h - 2$; ein Teilbaum hat jedoch die Höhe $h - 1$. Nach Definition sind T_l und T_r wieder AVL – Bäume. Damit ergibt sich:

$$\begin{aligned} \|T\| &= \|T_l\| + \|T_r\| + 1 \\ &\stackrel{(IV)}{\geq} \left(\frac{1+\sqrt{5}}{2}\right)^{h-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{h-2} + 1 \\ &\geq \left(\frac{1+\sqrt{5}}{2}\right)^{h-2} \left(\frac{1+\sqrt{5}}{2} + 1\right) \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^{h-2} \left(\frac{1+\sqrt{5}}{2}\right)^2 \\ &= \left(\frac{1+\sqrt{5}}{2}\right)^h \end{aligned}$$

□

Damit ergibt sich unmittelbar: Ist ein geordnetes Wörterbuch der Größe n als AVL – Baum implementiert, so lässt sich **find** mit Zeitkomplexität $O(\log n)$ realisieren.

Begründung: Satz 8 impliziert $2n + 1 \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$, also

$$h \leq (\log(2n + 1)) \frac{1}{\log \frac{1+\sqrt{5}}{2}} = O(\log n)$$

Wir betrachten nun **insert**(o, k). Für einen binären Suchbaum T und einen Knoten x in T ist der BALANCIERUNGSINDEX $B(x)$ definiert als

$$B(x) =_{\text{def}} h(T_l(x)) - h(T_r(x))$$

wobei $T_l(x)$ bzw. $T_r(x)$ der linke bzw. rechte Teilbaum von x seien. Für AVL – Bäume gilt stets $-1 \leq B(x) \leq 1$.

Wir erweitern im Folgenden AVL – Bäume so, dass in jedem Knoten x immer die Höhe des Teilbaumes mit x als Wurzel abgespeichert wird. Dazu benötigen wir zusätzlich $O(\log \log n)$ Bits pro Knoten.

Beim Einfügen eines Schlüssels k wird zunächst wie bei binären Suchbäumen der Knoten gesucht, bei dem **find** erfolglos abbricht. An dieser Stelle wird ein neuer Schlüssel mit Inhalt k eingefügt (mit Blättern). Die Höhenbalancierung wird durch Einfügen nur dann verletzt, wenn:

- in Knoten x mit $B(x) = 1$ $T_l(x)$ um ein Level höher wird oder analog
- in Knoten x mit $B(x) = -1$ $T_r(x)$ um ein Level höher wird.

Es sei x der Knoten in T , bei dem auf dem Pfad vom neu eingefügten Knoten zur Wurzel das erste Mal die Höhenbedingung verletzt wird. Damit gibt es im Wesentlichen die zwei Fälle aus Abbildung 3.1. (Weitere Fälle entstehen durch Vertauschung von links und rechts; diese können aber analog behandelt werden.)

Durch Umhängen (Rotation) von Teilbäumen kann die Höhenbalancierung wieder hergestellt werden — vgl. Abb. 3.2. Damit: in AVL – Bäumen lässt sich **insert** mit Zeitkomplexität $O(\log n)$ realisieren.

Begründung:

- Bestimmen der Einfügeposition kosten $O(h(T))$ Schritte.

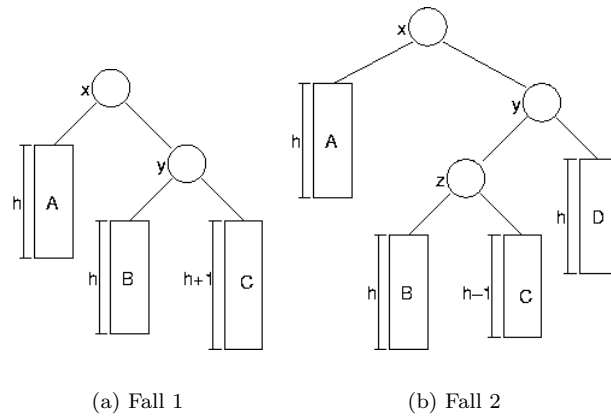


Abbildung 3.1: Situationen, die beim Einfügen ein Balancieren erfordern

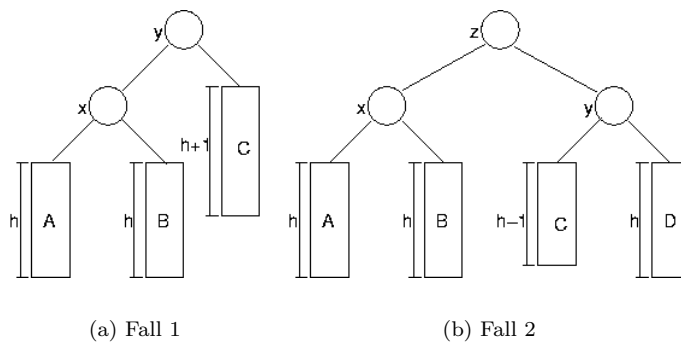


Abbildung 3.2: Ausbalancierte Bäume nach dem Einfügen

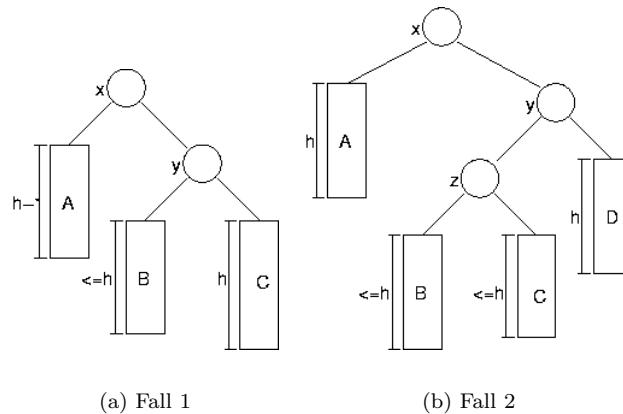


Abbildung 3.3: Situationen, die beim Löschen ein Balancieren erfordern

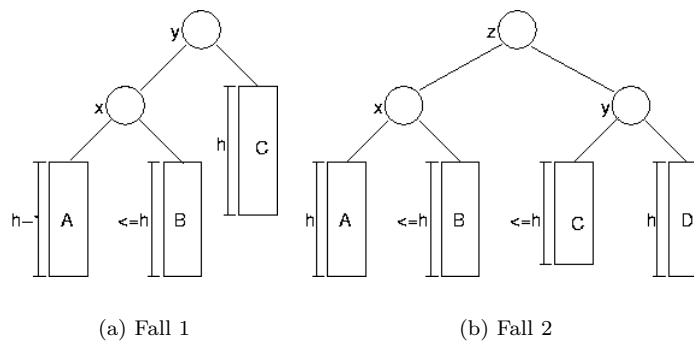


Abbildung 3.4: Ausbalancierte Bäume nach dem Löschen

- Bestimmen, ob und wo Höhenbalancierung gestört wird, kostet $O(h(T))$ Schritte.
- Rotation (auch Doppelrotation) kostet $O(1)$ Schritte.

⇒ Insgesamt Zeitkomplexität $O(h(T))$, d.h. $O(\log n)$ nach Satz 8.

Jetzt: `delete(k)`

Wir gehen zunächst genauso vor wie bei binären Suchbäumen üblich, d.h. nach Vertauschung von Schlüssel k mit nächstgrößeren Schlüssel im Baum wird nur Knoten mit maximal einem inneren Knoten als Kind gelöscht. Die Höhenbalancierung wird durch Löschen nur dann verletzt, wenn

- in Knoten x mit $B(x) = 1$ wird $T_r(x)$ um ein Level kleiner
- in Knoten x mit $B(x) = -1$ wird $T_l(x)$ um ein Level kleiner

O.B.d.A. betrachten wir nur den zweiten Fall. Es sei x der Knoten, bei dem zum ersten Mal (auf dem Weg vom gelöschten Knoten zur Wurzel) die Höhenbalancierung verletzt wird. Dabei gibt es im Wesentlichen wieder zwei Fälle, dargestellt in Abbildung 3.3. (Weitere Fälle durch Vertauschung.)

Durch Umhängen (Rotation) von Teilbäumen ergibt sich die in Abbildung 3.4 dargestellte Situation. Damit: in AVL – Bäumen lässt sich `delete` mit Zeitkomplexität $O(\log n)$ realisieren. *Begründung:* klar (wie bei `delert`).

3.3 (a, b) – Bäume

(1982 von Huddleston und Mehlhorn eingeführt)

Idee: Innere Knoten können größeren Grad haben und Tupel speichern; Einträge sagen uns, in

welchem Teilbaum wir weitersuchen müssen.

Wir betrachten dazu allgemein Suchbäume:

- Objekte werden nur in Blättern abgespeichert (in jedem Blatt nur ein Objekt).
- Hat ein innerer Knoten m Kinder, so enthält er $m - 1$ Schlüssel.
- Die Schlüssel im i – ten Teilbaum sind kleiner als die Schlüssel im $(i + 1)$ – ten Teilbaum.
- Enthaltene Knoten die Schlüssel $k_1 < k_2 < \dots < k_{m-1}$. Dann kann ein Objekt mit Schlüssel $k \in (k_{i-1}, k_i]$ (mit $k_0 = -\infty$ und $k_m = \infty$) nur im i – ten Teilbaum enthalten sein.

Es seien $a, b \in \mathbb{N}$ mit $a \leq 2$ und $b \leq 2a - 1$. Ein allgemeiner Suchbaum heißt (a, b) – BAUM genau dann, wenn folgendes gilt:

- Alle Blätter befinden sich auf gleichem Level.
- Jeder innere Knoten hat maximal b Kinder.
- Die Wurzel hat mindestens zwei Kinder.
- Jeder innere Knoten, der nicht Wurzel ist, hat mindestens a Kinder.

Satz 9 Ein (a, b) – Baum mit n Blättern hat mindestens die Höhe $\lceil \log_b n \rceil$ und maximal die Höhe $1 + \lfloor \log_a \frac{n}{2} \rfloor$.

Beweis. Maximale Anzahl von Blättern in einem (a, b) – Baum der Höhe h : b^h

$$\Rightarrow h \leq b^h \Rightarrow \lceil \log_b n \rceil \leq h$$

Minimale Anzahl von Blättern in einem (a, b) – Baum der Höhe h : $2a^{h-1}$

$$\Rightarrow n \geq 2a^{h-1} \Rightarrow \log_a \frac{n}{2} \geq h - 1 \Rightarrow 1 + \lfloor \log_a \frac{n}{2} \rfloor \geq h$$

□

- **find(k)**: Bestimme rekursiv über die inneren Knoten den jeweils nächsten gültigen Teilbaum, bis die Suche im Blatt erfolgreich bzw. erfolglos endet; verwende in inneren Knoten lineare Suche. Damit: Komplexität von **find** ist $O(\log n)$. Begründung: Nach Satz 9 müssen maximal $O(\log_a n)$ innere Knoten überprüft werden. Lineare Suche: $O(b)$ Vergleiche (bei festem b) $\Rightarrow O(b \log_a n) = O(\log_a n)$ Schritte.
- **insert(o, k)**: Wie bei Suchbäumen üblich gelangen wir nach erfolgloser Suche in ein Blatt y mit $k \neq \text{key}(y)$. O.B.d.A. sei $k < \text{key}(y)$. Sei x Elternknoten von y (x ist innerer Knoten). Dann gibt es Schlüssel k_{i-1}, k_i , die in x gespeichert sind mit $k_{i-1} < k < \text{key}(y) \leq k_i$. Hänge neues Blatt \hat{y} mit Schlüssel k links von y in x ein. Zwei Fälle sind möglich:
 1. Knoten x hat immer noch $\leq b$ Kinder \Rightarrow fertig.
 2. Knoten x hat nun genau $b + 1$ Kinder; dann verfähre wie folgt: Da $b + 1 \leq 2a$ ist, teilen wir x in zwei Knoten mit jeweils $\geq a$ Kindern auf und verschieben den überschüssigen Schlüssel in den Elternknoten von x . Dies erhöht die Anzahl der Kinder von **parent(x)** um eins; führe nun das Verfahren rekursiv weiter, bis es abbricht oder die Wurzel erreicht. Damit: Komplexität von **insert** ist $O(\log n)$ (da maximal $O(\log n)$ Knoten entlang eines Pfades vom Blatt bis zur Wurzel betrachtet werden müssen).
- **delete(k)**: Nach erfolgreicher Suche sind wir in einem Blatt mit Schlüssel k gelandet und entfernen dieses Blatt mit zugehörigem Schlüssel im Elternknoten x . Zwei Fälle sind möglich:
 1. Knoten x hat immer noch $\geq a$ Kinder \Rightarrow fertig.
 2. Knoten x hat nun $a - 1$ Kinder; dann verfähre wie folgt: Verschmelze x mit einem seiner Nachbarn zu einem neuen Knoten \hat{x} . Hat \hat{x} nun mindestens $b + 1 \geq 2a$ Kinder, so spalte \hat{x} in zwei Knoten mit jeweils $\geq a$ Kindern auf und wir sind fertig. Hat \hat{x} nun $\leq b$ Kinder, so ist die Anzahl der Kinder für **parent(x)** gesunken; führe nun das Verfahren rekursiv fort, bis es abbricht oder die Wurzel erreicht. Damit: Komplexität von **delete** ist $O(\log n)$ (analog zu **insert**).

3.4 Splay Trees

- 1985 von Sleator und Tarjan eingeführt
- »selbstorganisierende binäre Suchbäume«
- besitzen keine expliziten Regeln zur »globalen« Höhenbalancierung, aber stattdessen »lokale« Operationen, die `find`, `insert` und `delete` mit amortisierter Komplexität $O(\log n)$ garantieren

Ein Splay Tree T ist ein binärer Suchbaum, auf dem die Operationen `zig-zig` (Abbildung 3.5), `zig-zag` (Abbildung 3.6) und `zig` (Abbildung 3.7) ausgeführt werden. Es sei k ein Schlüssel, der

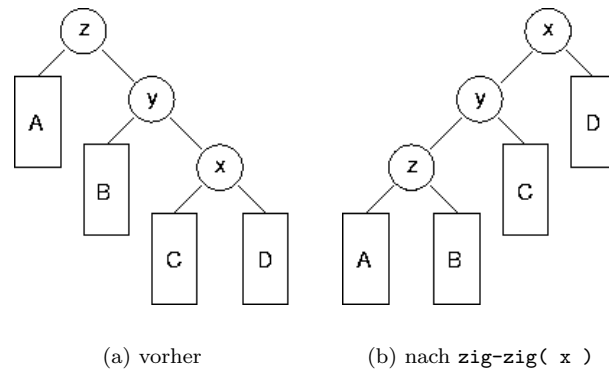


Abbildung 3.5: `zig-zig`

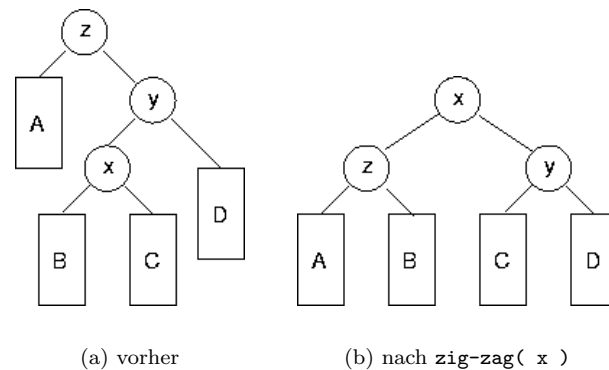


Abbildung 3.6: `zig-zag`

im Splay Tree T vorkommt, wobei alle Schlüssel in T verschieden sind. Ein `SPLAYING` von k ist eine minimale Folge von Splay – Operationen, die jeweils auf den k enthaltenden Knoten angewendet werden, so dass nach Abarbeitung der Folge k in der Wurzel des Baumes steht.

Beachte: Splaying ist nicht eindeutig.

Komplexität eines Splayings (d : Tiefe des Knotens, der k enthält):

- Splay – Operationen in $O(1)$ Schritten
- `zig-zig`, `zig-zag` reduzieren die Tiefe von k um 2
- `zig` reduziert die Tiefe von k um 1

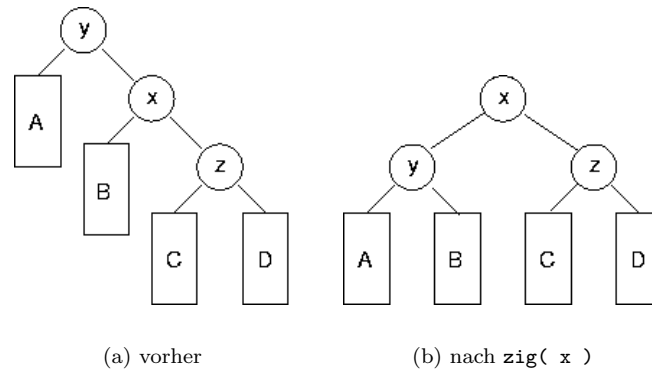


Abbildung 3.7: zig

Damit: Splaying von k benötigt $\lfloor \frac{d}{2} \rfloor$ zig-zig – oder zig-zag – Operationen und ein zig, falls d ungerade.

\Rightarrow Splaying in $\Theta(d)$ Schritten möglich. (Beachte: Dies entspricht auch der Komplexität von $\text{find}(k)$.)

Wann wird Splaying ausgeführt?

- $\text{find}(k)$: Bei erfolgreicher Suche führe Splaying von k aus. Bei erfolgloser Suche führe Splaying des Schlüssels im Elternknoten des Blattes aus, in dem die Suche endete.
- $\text{insert}(o, k)$: Führe Splaying des neu eingefügten Schlüssels aus.
- $\text{delete}(k)$: Führe Splaying des Schlüssels im Elternknoten des gelöschten Knotens aus.

Ansonsten: find , insert , delete wie bei binären Suchbäumen üblich.

Komplexität von find , insert , delete : $O(h(T))$ im worst case, d.h. $O(n)$ im schlechtesten Fall (z.B. Objekte in aufsteigender Schlüsselreihenfolge eingefügt;) dann gilt $h(T) = n$.

Amortisierte Analyse

Wir betrachten eine Folge von m Operationen find , insert und delete . Es sei T ein Splay Tree mit n Schlüsseln. Für einen Knoten $v \in T$ definieren wir:

$$\begin{aligned} S(v) &=_{\text{def}} \|T_v\| && \text{(Gewicht von } v\text{)} \\ R(v) &=_{\text{def}} \log S(v) && \text{(Rang von } v\text{)} \\ R(T) &=_{\text{def}} \sum_{v \in T} R(v) && \text{(Rang von } T\text{)} \end{aligned}$$

Mit \hat{S} , \hat{R} werden wir die Gewichte und Ränge nach Ausführung von Splay – Operationen bezeichnen.

Lemma 10 Es sei T ein Splay Tree und x ein Knoten in T . Dann gilt:

1. $\hat{R}(T) - R(T) \leq 3(\hat{R}(x) - R(x)) - 2$ für zig-zig und zig-zag auf x ,
2. $\hat{R}(T) - R(T) \leq 3(\hat{R}(x) - R(x))$ für zig auf x .

Beweis. Es gilt:

$$S(x) = 1 + S(\text{leftChild}(x)) + S(\text{rightChild}(x))$$

- zig-zig: Es sind nur $x, y = \text{parent}(x)$ und $z = \text{parent}(y)$ relevant. Weiterhin gilt:

1. $\hat{R}(x) = R(z)$

2. $\hat{R}(y) \leq \hat{R}(x)$
3. $R(y) \geq R(x)$
4. $\hat{R}(z) \leq 2\hat{R}(x) - R(x) - 2$

Begründung für (4): Es gilt: $S(x) + \hat{S}(z) \leq \hat{S}(x)$. Damit:

$$\begin{aligned}
R(x) + \hat{R}(z) &= \log S(x) + \log \hat{S}(z) \\
&= \log S(x) \cdot \hat{S}(z) \\
&\stackrel{a,b \geq 0 \Rightarrow ab \leq \frac{(a+b)^2}{4}}{\leq} \log \frac{(S(x) + \hat{S}(z))^2}{4} \\
&\leq \log \frac{\hat{S}(x)^2}{4} \\
&= 2 \log \hat{S}(x) - 2 \\
&= 2\hat{R}(x) - 2
\end{aligned}$$

Insgesamt:

$$\begin{aligned}
\hat{R}(T) - R(T) &= \hat{R}(x) + \hat{R}(y) + \hat{R}(z) - R(x) - R(y) - R(z) \\
&\leq \hat{R}(x) + \hat{R}(x) + 2\hat{R}(x) - R(x) - 2 - R(x) - R(x) - \hat{R}(x) \\
&= 3\hat{R}(x) - 3R(x) - 2 \\
&= 3(\hat{R}(x) - R(x)) - 2
\end{aligned}$$

- **zig-zag:** Es sind nur $x, y = \text{parent}(x)$ und $z = \text{parent}(y)$ relevant. Weiterhin gilt:

1. $\hat{R}(x) = R(z)$
2. $R(x) \leq R(y)$
3. $\hat{R}(y) + \hat{R}(z) \leq 2\hat{R}(x) - 2$

Begründung zu (3): Es gilt: $\hat{S}(z) + \hat{S}(y) \leq \hat{S}(x)$. Damit:

$$\begin{aligned}
\hat{R}(z) + \hat{R}(y) &\leq \log \frac{(\hat{S}(z) + \hat{S}(y))^2}{4} \\
&\leq \log \frac{\hat{S}(x)^2}{4} \\
&= 2 \log \hat{S}(x) - 2 \\
&= 2\hat{R}(x) - 2
\end{aligned}$$

Insgesamt:

$$\begin{aligned}
\hat{R}(T) - R(T) &= \hat{R}(x) + \hat{R}(y) + \hat{R}(z) - R(x) - R(y) - R(z) \\
&\leq \hat{R}(x) + 2\hat{R}(x) - 2 - R(x) - R(x) - \hat{R}(x) \\
&= 2(\hat{R}(x) - R(x)) - 2 \\
&\leq 3(\hat{R}(x) - R(x)) - 2
\end{aligned}$$

- **zig:** Es sind nur $x, y = \text{parent}(x)$ relevant. Damit:

$$\begin{aligned}
\hat{R}(T) - R(T) &= \hat{R}(x) + \hat{R}(y) - R(x) - R(y) \\
&\leq \hat{R}(x) + \hat{R}(x) - R(x) - R(x) \\
&= 2(\hat{R}(x) - R(x)) \\
&\leq 3(\hat{R}(x) - R(x))
\end{aligned}$$

□

Lemma 11 Es seien T ein Splay Tree mit Wurzel t , x ein Knoten von T der Tiefe d , auf dem ein Splaying ausgeführt wird. Dann gilt für den Rang \hat{R} nach dem Splaying:

$$\hat{R}(T) - R(T) \leq 3(R(t) - R(x)) - d + 2$$

Beweis. Ein Splaying von x besteht aus $\lceil \frac{d}{2} \rceil$ Splay – Operationen. Definiere $p =_{\text{def}} \lceil \frac{d}{2} \rceil$. Es seien $R_i(x)$ die Ränge nach Ausführen der i -ten Splay – Operation, $i \in \{1, \dots, p\}$, $R_0(x) = R(x)$ und δ_i die Veränderungen von $R(T)$ nach Ausführen der i -ten Splay – Operation. Damit gilt:

$$\begin{aligned} \hat{R}(T) - R(T) &= \sum_{i=1}^p \delta_i \\ &= \sum_{i=1}^p (3(R_i(x) - R_{i-1}(x)) - 2) + 2 \\ &= 3(R_p(x) - R_0(x)) - 2p + 2 \\ &\leq 3(R(t) - R(x)) - d + 2 \end{aligned}$$

□

Satz 12 Für eine Folge von Operationen **find**, **insert** oder **delete** auf einem anfänglich leeren Splay Tree ist die Gesamtkomplexität der Ausführung

$$O(m \log n)$$

wobei n die Gesamtzahl aller **insert** – Operationen ist.

Beweis. (Amortisationsanalyse mit Bankkontomethode)

Wesentlicher Kostenfaktor sind Splay – Operationen, d.h. wir messen lediglich »Splaying – Zeit«.

- zig-zig, zig-zag kosten 2 Einheiten,
- zig kostet eine Einheit.

⇒ Splaying von Knoten der Tiefe d kostet d Einheiten.

Für das Gehalt G befolgen wir folgende Spar- und Ausgabenstrategie:

- Ist G gleich den Splaying – Kosten, dann verwende G vollständig für Splaying.
- Ist G größer als die Splaying – Kosten, dann verteile den Überschuss auf Sparkonten verschiedener Knoten.
- Ist G kleiner als die Splaying – Kosten, dann bezahle den Rest aus dem Guthaben von Sparkonten verschiedener Knoten.

Folgende Invariante muss aufrecht erhalten werden: »Vor und nach dem Splaying von Knoten x hat das Sparkonto von x ein Guthaben von mindestens x .«

Wir setzen $G =_{\text{def}} 3(R(t) - R(x)) + 2$ für jede Operation, wobei t die Wurzel des Splay Trees vor Ausführung der Operation ist und x der Knoten, auf dem ein Splaying innerhalb der Operation ausgeführt wird. Damit: Splaying von Knoten x der Tiefe d kostet d Einheiten und verursacht eine Änderung des Ranges von T von höchstens $3(R(t) - R(x)) - d - 2$ (nach Lemma 11).

⇒ G reicht aus, um das Splaying zu bezahlen und die Invariante aufrecht zu erhalten. Für jede Operation wird für genau einen Knoten ein Splaying ausgeführt. Außerdem gilt:

1. $R(x_i) \geq 1$
2. $R(t) \leq \log(2n + 1)$

Damit gilt für die Gesamtkomplexität:

$$\begin{aligned} &\leq \sum_{i=1}^m G = \sum_{i=1}^m (3(R(t) - R(x)) + 2) \\ &\leq 2m + 3m \log(2n + 1) \\ &= O(m \log n) \end{aligned}$$

□

3.5 Skiplisten

- 1990 von Pugh vorgeschlagen
- randomisierte Datenstruktur, die **find**, **insert** und **delete** in erwarteter Zeit $O(\log n)$ realisiert

Eine SKIPLISTE S für ein geordnetes Wörterbuch besteht aus einer Folge (S_0, S_1, \dots, S_n) von geordneten Listen mit folgenden Eigenschaften:

1. S_0 enthält alle Schlüssel (und zwei spezielle Schlüssel $-\infty$ und $+\infty$)
2. für alle $i \in \{0, \dots, h-1\}$ enthält S_{i+1} eine zufällige Auswahl von Schlüssel aus S_i (sowie $-\infty$ und $+\infty$)
3. S_k besteht nur aus $-\infty$ und $+\infty$

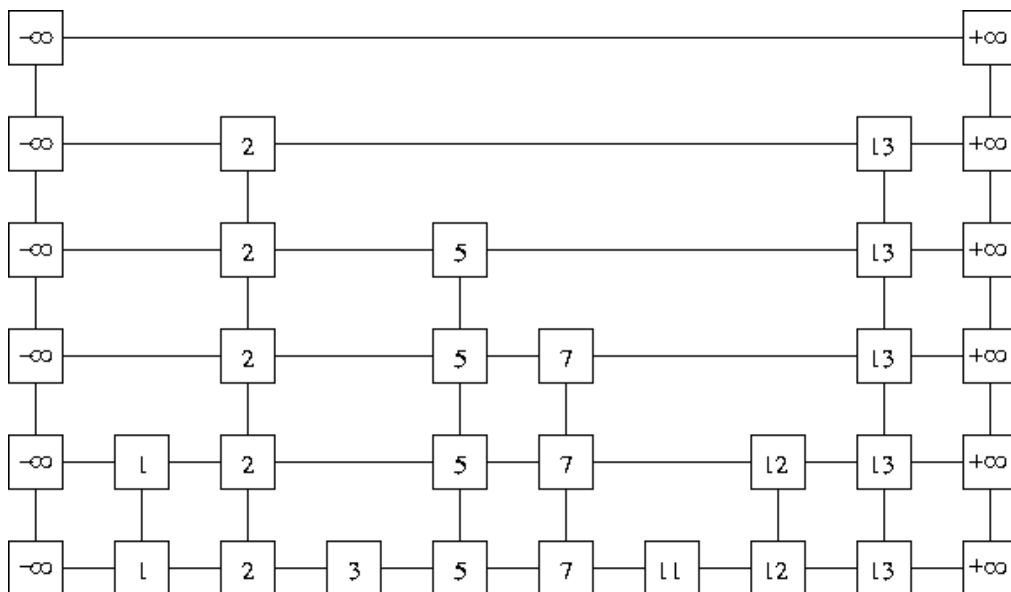


Abbildung 3.8: Skipliste

Horizontale Verknüpfungen sind Ebenen, vertikale Verknüpfungen sind Türme. Jeder Turm besteht aus identischen Schlüssel.

Traversierungsoperationen

(p Position in Skipliste)

- **after**(p): gibt die Position des nächstgrößeren Schlüssels nach p in der gleichen Ebene zurück
- **before**(p): gibt die Position des nächstkleineren Schlüssels nach p in der gleichen Ebene zurück
- **below**(p): gibt die Position des Schlüssels von p in der nächsttieferen Ebene des gleichen Turms zurück
- **above**(p): gibt die Position des Schlüssels von p in der nächsthöheren Ebene des gleichen Turms zurück

Komplexität der Operationen: $O(1)$ bei Implementierung mit doppelt verketteten Listen.

Wörterbuchoperationen

1. `find(k)`: Wir verwenden folgenden Suchalgorithmus `skipsearch`:
Eingabe: Schlüssel `k` Ausgabe: Position mit größtem Schlüssel $\leq k$

```

1  sei p die hoechstliegende Position von -UNENDLICH in S
2  WHILE below( p ) != NIL
3    p := below( p )
4    WHILE key( after( p ) ) <= k
5      p := after( p )
6  RETURN p

```

Damit für `find(k)`: Bestimme `p := skipsearch(k)` und teste, ob `k = key(p)`.

2. `insert(o, k)`: Bestimme Einfügeposition für Schlüssel `k`; füge `(o, k)` in S_0 ein und entscheide iterativ, wie hoch der Turm für Schlüssel `k` wird.
Algorithmus `insert(o, k)`

```

1  p := skipsearch( k )
2  q := NIL
3  REPEAT
4    q := insertAfterAbove( p, q, k )
5    WHILE above( p ) = NIL
6      p := before( p )
7      p := above( p )
8      random x in { 0, 1 }
9  until x = 0

```

`insertAfterAbove(p, q, k)` fügt dabei den Schlüssel `k` an der Position nach `p` über `q` ein und gibt die neue Position zurück.

3. `delete(k)`: Bestimme Posiiton von `k` und entferne `p` aus S_0 und alle Positionen oberhalb von `p`

Bemerkungen

1. `insert` und `delete` können ohne `above`- und `before`-Methode ausgeführt werden in Top-Down-Links-Rechts-Richtung.
2. Durch Einfügen könnte die Höhe der Skipliste wachsen; wir müssen also die Referenzierung auf das höchstliegende $-\infty$ aktualisieren.
3. Die Höhe der Skipliste könnte auch beschränkt werden (in Abhängigkeit von n); z.B. ist $h = 3\lceil \log n \rceil$ eine gute Wahl.

Komplexitätsanalyse

Worst – Case – Verhalten: Wird die Höhe nicht beschränkt, so könnte die Liste unbeschränkt wachsen (falls genügend Platz vorhanden wäre). Bei beschränkter Höhe h ist die Komplexität von `find`, `insert` und `delete` $O(n+h)$ im schlechtesten Fall (d.h., wenn alle Türme Höhe $h+1$ haben). *Aber*: Schlechteste Fälle haben vernachlässigbare Wahrscheinlichkeit des Eintretens.

Satz 13 Für die Höhe h einer Skipliste zu einem geordneten Wörterbuch der Größe n gilt $h = O(\log n)$ mit Wahrscheinlichkeit $1 - o(1)$.

Beweis. Betrachte die Zufallsvariable $X_i : \{1, \dots, n\} \rightarrow \{0, 1\}$ mit

$$X_i(k_j) = \begin{cases} 1 & \text{falls Schlüssel } k_j \text{ in } S_i \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Es gilt $\mathcal{P}(X_i(k_j) = 1) = \frac{1}{2^i}$. Damit gilt für $\hat{x}_i = \sum_{j=1}^n X_i(k_j)$ (Anzahl der Schlüssel in S_i):

$$\mathcal{P}(\hat{X}_i \geq 1) \leq \sum_{j=1}^n \mathcal{P}(X_i(k_j) = 1) = \frac{n}{2^i}$$

Daraus folgt:

$$\mathcal{P}(\hat{X}_{\lceil c \log n \rceil} \geq 1) \leq \frac{n}{2^{\lceil c \log n \rceil}} \leq \frac{n}{2^{c \log n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

Somit:

$$\mathcal{P}(h < \lceil c \log n \rceil) = 1 - \mathcal{P}(\hat{X}_{\lceil c \log n \rceil} \geq 1) \geq 1 - \frac{1}{n^{c-1}}$$

□

Erwartete Laufzeit der Operationen

- find:** Wir müssen zwei WHILE – Schleifen berücksichtigen; die äußere ist durch die erwartete Höhe bestimmt ($O(\log n)$ nach Satz 13). Für die innere Schleife müssen die Elemente gezählt werden, die bei Wechsel von höherer zur nächsttieferen Ebene zusätzlich überprüft werden. Es sei n_i die Anzahl der Schlüssel (ohne $-\infty, +\infty$), die auf Ebene i zusätzlich überprüft werden müssen, d.h. die Wahrscheinlichkeit, dass Schlüssel aus S_i nicht zu S_{i+1} gehört und damit für n_i gezählt wird, ist $\frac{1}{2}$.
 \Rightarrow Im Mittel müssen auf jeder Ebene lediglich zwei Schlüssel zusätzlich behandelt werden (Erwartungswert der geometrischen Verteilung für $p = \frac{1}{2}$).
 \Rightarrow **find** in $O(\log n)$ erwarteter Laufzeit.
- insert:** $O(\log n)$ erwartete Laufzeit (mit ähnlicher Analyse wie bei **find**).
- delete:** $O(\log n)$ erwartete Laufzeit.

Platzbedarf von Skiplisten

Erwartete Anzahl von Schlüsseln in einer Skipliste der Höhe h :

$$\leq \sum_{j=0}^h \frac{n}{2^j} = n \sum_{j=0}^h \frac{1}{2^j} < 2n$$

$\Rightarrow O(n)$ erwarteter Platzbedarf.

Zusammenfassung der Datenstrukturen

	find	insert	delete	Platz	
Binäre Suchbäume	$O(n)$	$O(1)$	$O(n)$	$O(n)$	
AVL-Bäume	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	
(a, b) – Bäume	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	
Splay Trees	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	(amortisiert)
Skiplisten	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	(erwartet)

Kapitel 4

Sortieren und selektieren in Mengen

Wir erweitern geordnete Wörterbücher um weitere Operationen (z.B. Vereinigung, Mengendifferenz, Medianbestimmung etc.). Für die Implementierung ist die Aufrechterhaltung einer Sortierreihenfolge ein probates Mittel.

Sortierproblem

Gegeben seien Objekte o_1, \dots, o_n , wobei jedes Objekt o_i einen (eindeutigen) Schlüssel k_i besitzt. Das Universum der Schlüssel ist total geordnet. Finde eine Reihenfolge o_{i_1}, \dots, o_{i_n} , so dass für die zugehörigen Schlüssel $k_{i_1} \leq \dots \leq k_{i_n}$ gilt.

4.1 Analyse von Quicksort

Quicksort ist eines der schnellsten und einfachsten Sortierverfahren. Es basiert auf dem Teile-und-Herrsche-Ansatz.

Algorithmus: Quicksort

- Eingabe: Array A (mit n Elementen); Parameter $l, r \in \{1, \dots, n\}$
- Ausgabe: Array A mit $A[l] \leq \dots \leq A[r]$

```
1  IF l < r
2    i := l;
3    j := r - 1;
4    WHILE i <= j
5      WHILE A[i] <= A[r] AND i <= j
6        i := i + 1;
7      WHILE A[j] >= A[r] AND i < j
8        j := j - 1;
9      IF i < j
10     A[i], A[j] := A[j], A[i];
11     i := i + 1;
12     j := j - 1;
13  A[l], A[r] := A[r], A[l];
14  Quicksort( A, l, i - 1 );
15  Quicksort( A, i + 1, r );
```

Erstaufruf: Quicksort(A, 1, n);
Element A[r] heißt Pivot-Element.

Es sei A eine total geordnete Menge mit n Elementen. Für $x \in A$ ist der RANG von x in A definiert als

$$\text{rank}_A(x) =_{\text{def}} \|\{z \in A : z < x < j\}\| + 1$$

Ein Element $x \in A$ heißt MEDIAN, falls $\text{rank}_A(x) = \lceil \frac{n+1}{2} \rceil$.

Beachte: Alle Elemente von A verschieden.

Komplexität (gemessen in Anzahl der Vergleiche von Feldelementen):

- Zeilen 2 bis 13 benötigen $r - 1$ Vergleiche.
- Rekursion:

$$\begin{aligned} V(1) &= 0 \\ V(n) &= n - 1 + V(k - 1) + V(n - k) \end{aligned}$$

für ein $k \in 1, \dots, n$, $n \geq 2$, wobei k der Rang des Pivot-Elementes ist.

Damit:

- im schlechtesten Fall (sortierte Folge) $\Theta(n^2)$ Vergleiche.
- im besten Fall (Rang des Pivot-Elementes ist immer der Median) $O(n \log n)$ Vergleiche.

Nun betrachten wir die randomisierte Version von Quicksort und fügen dazu folgende Befehlszeilen zwischen Zeile 1 und 2 ein:

```

1      ...
2      RANDOM x in { 1, ..., r };      # Zeile 1a
3      A[x], A[r] := A[r], A[x];      # Zeile 1b
4      ...

```

Wir wollen nun die Komplexität des randomisierten Quicksort in Abhängigkeit von der Qualität des Zufallszahlengenerators analysieren. Es sei $V_\pi(n)$ die erwartete Anzahl von Vergleichen von Quicksort auf dem Array $A[1 \dots n]$, dessen Elemente gemäß $\pi \in S_n$ permutiert sind, d.h. $A[\pi(1)] < \dots < A[\pi(n)]$. Weiterhin sei X eine Zufallsvariable mit Werten in $\{1, \dots, n\}$, d.h. X modelliert einen Zufallszahlengenerator in Zeile 1a.

Wir interessieren uns für die erwartete Anzahl von Vergleichen im schlechtesten Fall, d.h. für $V(n) = \max_{\pi \in S_n} V_\pi(n)$. Es gilt $V(1) = 0$. Für $n \geq 2$ gilt

$$\begin{aligned} V(n) &= \max_{\pi \in S_n} V_\pi(n) \\ &= (n - 1) + \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) (V_{\pi^1}(i - 1) + V_{\pi^2}(n - i)) \\ &\leq (n - 1) + \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) \left(\max_{\varphi \in S_{i-1}} V_\varphi(i - 1) + \max_{\varphi \in S_{n-i}} V_\varphi(n - i) \right) \\ &= (n - 1) + \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) (V(i - 1) + V(n - i)) \quad (*) \end{aligned}$$

Hierbei ist $p_\pi(i)$ die Wahrscheinlichkeit dafür, dass das gewählte Pivot-Element den Rang i hat, d.h. $p_\pi(i) = \Pi[\pi(x) = i]$.

Beispiele

- $X = n$, dann $p_\pi(i) = 0$ für $1 \leq i \leq n - 1$, $p_\pi(n) = 1$ für alle $\Pi \in S_n$ mit $\Pi(n) = n$.
- X gleichverteilt und unabhängig, dann $p_\pi(i) = \frac{1}{n}$ für alle $i \in \{1, \dots, n\}$ und $\pi \in S_n$.

Der Unterschied zwischen Verteilungen wird durch die (binäre/Shannon) Entropie gemessen:

$$H(x) =_{\text{def}} -x \log_2(x) - (1 - x) \log_2(1 - x) \quad \text{für } x \in [0, 1]$$

Wesentlich:

- $H\left(\frac{1}{2}\right) = 1$
- $H\left(\frac{1}{2} + z\right) = H\left(\frac{1}{2} - z\right)$
- $H(n) \leq H(y)$ für $0 \leq x \leq y \leq \frac{1}{2}$.

Satz 14 Für jede monoton wachsende Funktion g mit der Eigenschaft

$$g(n) \geq \left(\min_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) H\left(\frac{i}{n}\right) \right)^{-1}$$

gilt

$$V(n) \leq g(n)n \log_2(n)$$

Beweis. (Induktion über n)

(IA) Für $n \leq 1$: $V(1) = 0$.

(IS) Es sei $n \geq 2$. Damit erhalten wir aus (*):

$$\begin{aligned} V(n) &\leq V(n-1) + \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i)(V(i-1) + V(n-i)) \\ &\leq (n-1) + \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i)(g(i-1)(i-1) \log(i-1) + g(n-i)(n-i) \log(n-i)) \\ &\leq n + ng(n) \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) \left(\frac{i}{n} \log i + \left(1 - \frac{i}{n}\right) \log(n-i) \right) \\ &= n + ng(n) \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) \left(\frac{i}{n} \log \frac{i}{n} + \left(1 - \frac{i}{n}\right) \log \left(1 - \frac{i}{n}\right) + \frac{i}{n} \log n + \left(1 - \frac{i}{n}\right) \log n \right) \\ &= n + ng(n) \log n + ng(n) \max_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) \left(\frac{i}{n} \log \frac{i}{n} + \left(1 - \frac{i}{n}\right) \log \left(1 - \frac{i}{n}\right) \right) \\ &= n + ng(n) \log n - ng(n) \min_{\pi \in S_n} \sum_{i=1}^n p_\pi(i) H\left(\frac{i}{n}\right) \\ &\leq g(n)n \log n \end{aligned}$$

□

Für eine perfekte Zufallsquelle, d.h. für $p_\pi(i) = \frac{1}{n}$ für alle $i \in \{1, \dots, n\}$ und $\pi \in S_n$, ergibt sich

$$g(n) = \left(\frac{1}{n} \sum_{i=1}^n H\left(\frac{i}{n}\right) \right)^{-1} \approx \left(\int_0^1 H(x) dx \right)^{-1}$$

Es gilt:

$$\begin{aligned} \int_0^1 H(x) dx &= - \int_0^1 x \log x dx - \int_0^1 (1-x) \log(1-x) dx \\ &= \frac{1}{\ln 2} \left(- \int_0^1 x \ln x dx - \int_0^1 (1-x) \ln(1-x) dx \right) \\ &= \frac{-2}{\ln 2} \int_0^1 x \ln x dx \\ &= \frac{-2}{\ln 2} \left[\frac{1}{2} x^2 \ln x - \frac{1}{4} x^2 \right]_0^1 \\ &= \frac{-2}{\ln 2} \left(-\frac{1}{4} \right) = \frac{1}{2 \ln 2} \end{aligned}$$

Korollar 15 Randomisiertes Quicksort mit perfekter Zufallsquelle benötigt im Erwartungswert max. $(2 \ln 2)n \log n \approx 1.38n \log n$ Vergleiche zum Sortieren von Elementen.

Beispiel für »beschränkte« Entropie: Es sei $t = o(n)$ eine monoton steigende Funktion. Definiere

$$p_\pi(i) =_{\text{def}} \begin{cases} \frac{1}{t(n)} & \text{falls } i \leq \frac{1}{2}t(n) \\ \frac{1}{t(n)} & \text{falls } i > n - \frac{1}{2}t(n) \\ 0 & \text{sonst} \end{cases}$$

D.h. wir wählen die Pivotelemente zufällig gleichverteilt unter den $t(n)$ schlechtesten Elementen von A .

Beachte: Die Wahrscheinlichkeiten hängen nicht von Π ab.

Wir müsse $\sum_{i=1}^n p_\pi(i) H\left(\frac{1}{2}\right)$ nach unten abschätzen:

$$\begin{aligned} \sum_{i=1}^n p_\pi(i) H\left(\frac{i}{n}\right) &= 2 \sum_{i=1}^{\frac{1}{2}t(n)} \frac{1}{t(n)} H\left(\frac{i}{n}\right) \\ &= \frac{2}{t(n)} \sum_{i=1}^{\frac{1}{2}t(n)} H\left(\frac{i}{n}\right) \\ &= \frac{2}{t(n)} \sum_{i=1}^{\frac{1}{2}t(n)} - \left(\frac{i}{n} \log \frac{i}{n} + \frac{n-i}{n} \log \frac{n-i}{n} \right) \\ &\geq \frac{2}{t(n)} \sum_{i=1}^{\frac{1}{2}t(n)} \frac{i}{n} \log \frac{n}{i} \\ &\geq \frac{2}{nt(n)} \log \frac{2n}{t(n)} \sum_{i=1}^{\frac{1}{2}t(n)} i \\ &= \frac{2}{nt(n)} \log \frac{2n}{t(n)} \frac{\frac{1}{2}t(n) (\frac{1}{2}t(n) + 1)}{2} \\ &\geq \frac{t(n)}{4n} \log \frac{2n}{t(n)} \end{aligned}$$

Aus Satz 14 folgt somit

$$V(n) \leq \frac{4n}{t(n)} \frac{1}{\log \frac{2n}{t(n)}} n \log n = \frac{4n^2}{t(n)} \frac{\log n}{\log \frac{2n}{t(n)}}$$

D.h. $t = \omega(1)$, so ist $V(n) = o(n^2)$.

Bemerkungen

1. Eine δ -Zufallsquelle produziert eine 1 als Ausgabe mit Wahrscheinlichkeit $\delta \leq p \leq 1 - \delta$. Das heißt, eine $\frac{1}{2}$ -Zufallsquelle ist eine perfekte Zufallsquelle.

Definiere einen Zufallsgenerator X auf dem Bereich $\{1, \dots, n\}$ durch $\lceil \log n \rceil$ Zufallsbits aus einer δ -Zufallsquelle, interpretiert als Zahl Y , und setze $X =_{\text{def}} (Y \bmod n) + 1$.

Dann gibt es für $0 < \delta < \frac{1}{2}$ ein $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ und $\pi \in S_n$ Satz 14 mit

$$g(n) = c_\delta \frac{1}{\sqrt{\log n}} n^{1-H(\delta)}$$

angewendet werden kann.

(D.h. $V(n) = O(n^{2-H(\delta)} \sqrt{\log n})$.)

2. Untere Schranken: Es gilt $V(n) \geq cg(n)n$ für geeignetes $c > 0$, falls g folgenden Bedingungen genügt:

- $g(n) \leq \left(\frac{n}{n-1} \sum_{i=1}^n p_{\pi}(i) H \left(\frac{i}{n+1} \right) \right)^{-1}$ und
- $\frac{g(i)}{g(n)} \geq \frac{i}{n}$ für $0 \leq i \leq n$

4.2 Selektionsalgorithmen

Problem: Gegeben eine (unsortierte) Folge von Schlüsseln und eine Zahl $k \in \mathbb{N}_+$. Bestimme das Element der Folge mit Rang k .

Vereinfachung: Alle Elemente sind paarweise verschieden.

BFPRT-Algorithmus

Der BFPRT-Algorithmus garantiert $O(n)$ Vergleiche. BFPRT steht für

- Manuel Blum
- Robert W. Floyd
- Vaughan R. Pratt
- Ronald L. Rivest
- Robert E. Tarjan

O.B.d.A sei $n > 800$ — sonst verwende Sortieralgorithmus.

Idee: Es gelte $5|n$. Teile n -Tupel in 5-Tupel auf und bestimme den Median jedes Tupels; bestimme nun rekursiv den Median der Menge aller Mediane (und gleichzeitig auch seinen Rang). Nun:

Nun:

- Rang vom Median der Mediane = $k \rightarrow$ fertig
- Rang vom Median der Mediane $> k \rightarrow$ entferne alle Elemente rechts ober vom Median der Mediane und suche im Rest Rang- k -Element.
- Rang vom Median der Mediane $< k \rightarrow$ entferne alle Elemente links unterhalb vom Median der Mediane und suche im Rest Rang- $(k - \text{entfernte Elemente})$ -Element.

Algorithmus BFPRT

- Eingabe: Array A, Parameter l, r, k (Rang)
- Ausgabe: Position des Rang- k -Elements in A (wobei A unsortiert werden kann)

Erstaufruf: BFPRT(A, 0, n-1, k)

```

1  IF ( r + 1 - l ) <= 800
2    HeapSort( A, l, r )
3    return k - l + 1
4  ELSE
5    c := ceil( ( r + 1 - l ) / 5 )
6    FOR i := 0 TO l - 1
7      j := 0
8      WHILE ( ( i + j * c ) <= ( r - l ) )
9        C[j] := A[l + i + j * c]
10     j := j + 1
11     HeapSort( C, 0, j - 1 )
12     FOR h := 0 TO j - 1
13       A[ l + i + h * c ] := C[h]
14     m := BFPRT( A, 2c, 3c - 1, ceil( ( l + 1 ) / 2 ) )
15     vertausche A[m] und A[r]
```

```

16   i := l - 1
17   j := r
18   WHILE ( i < j )
19     REPEAT
20       i := i + 1
21     UNTIL NOT ( ( i < j ) AND ( A[i] < A[r] ) )
22     REPEAT
23       j := j - 1
24     UNTIL NOT ( ( j > i ) AND ( A[j] > A[r] ) )
25     IF ( i >= 1 )
26       vertausche A[i] und A[r]
27     ELSE
28       vertausche A[i] und A[j]
29     IF ( k + 1 = i + 1 )
30       return i
31     IF ( k + 1 < i + 1 )
32       return BFPRT( A, l, i - 1, k )
33     IF ( k + 1 > i + 1 )
34       return BFPRT( A, i + 1, r, k - i )

```

Komplexität von BPRT

Zeile 11 mit Heapsort kann durch jede Strategie ersetzt werden, so dass eine beschriebene Feldkonfiguration, d.h. $C[0], C[1] < C[2] < C[3], C[4]$ mit maximal sieben Vergleichen gefunden werden kann.

Wir zeigen: BFPRT benötigt zur Bestimmung des Rang- k -Elementes aus n Elementen maximal $26n$ Vergleiche von Feldelementen (bzw. $O(n)$ Vergleiche). O.B.d.A. sei der Rang des Medians der Mediane größer als k für eine gegebene Eingabe, d.h. für $n > 800$ wird Zeile 32 ausgeführt. Damit wird der Feldbereich $N = r + 1 - l$ um mindestens

$$3 \left\lceil \frac{\lceil \frac{N}{5} \rceil}{2} \right\rceil - 4 \geq \frac{3}{10}N - 4$$

verkleinert. Es sei m die Größe des Tupels im rekursiven Aufruf. Dann gilt

$$m \leq N - \left(\frac{3}{10}N - 4 \right) = \frac{7}{10}N + 4$$

Bei Verwendung von Heapsort benötigen wir $2n \log n + \frac{13}{2}n$. Wir erhalten folgende Rekursionsgleichung:

$$V(n) = V\left(\left\lceil \frac{n}{5} \right\rceil\right) + V(m) + 7 \left\lceil \frac{n}{5} \right\rceil + n - 1$$

Wir zeigen nun $V(n) \leq 26n$ (mittels Induktion über n).

IA $n \leq 800$: Heapsort wird ausgeführt. Damit:

$$\begin{aligned} V(n) &\leq 2n \log n + \frac{13}{2}n \\ &\leq \frac{39}{2}n + \frac{13}{2}n = 26n \end{aligned}$$

IS $n > 800$: Rekursion wird ausgeführt. Damit:

$$\begin{aligned} V(n) &= V\left(\left\lceil \frac{n}{5} \right\rceil\right) + V(m) + 7 \left\lceil \frac{n}{5} \right\rceil + n - 1 \\ &\leq 26 \left\lceil \frac{n}{5} \right\rceil + 26m + 7 \left\lceil \frac{n}{5} \right\rceil + n - 1 \\ &\leq 26 \left(\frac{n}{5} + \frac{4}{5} \right) + 26 \left(\frac{7}{10}n + 4 \right) + 7 \left(\frac{n}{5} + \frac{4}{5} \right) + n - 1 \\ &\leq \frac{258n + 1264}{10} \leq 26n \end{aligned}$$

Bemerkungen:

- Der konstante Faktor ist nicht optimal (beste bekannte obere Schranke: $2.95n + o(n)$).
- $2n - o(n)$ Vergleiche sind im schlechtesten Fall immer notwendig.
- Quicksort mit Pivotisierung durch Medianbestimmung auf BFPRT-Basis benötigt $O(n \log n)$ Vergleiche.

4.3 Höhere Heapstrukturen: Binomial Queues

Wiederholung: Priority Queue mit Operationen:

- `insert(o, k)`
- `extractMin`
- `(minKey)`

Wir erweitern die Priority Queue um folgende Operationen:

- `find(k)`
- `delete(k)`
- `decreaseKey(k, k')`: setzt den Schlüsselwert des Objektes mit Schlüssel k auf k' .
- `merge(P, Q)`: verschmilzt die Priority Queues P und Q zu einer Priority Queue (unter der Annahme, dass P und Q disjunkte Schlüsselmengeten haben).

Binomial Queues basieren auf BINOMIALBÄUMEN, die gemäß Abbildung 4.1 definiert sind.

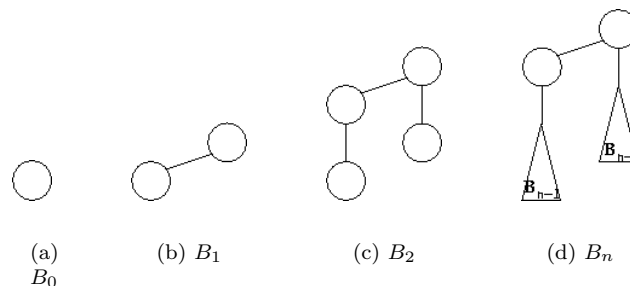


Abbildung 4.1: Binomialbäume

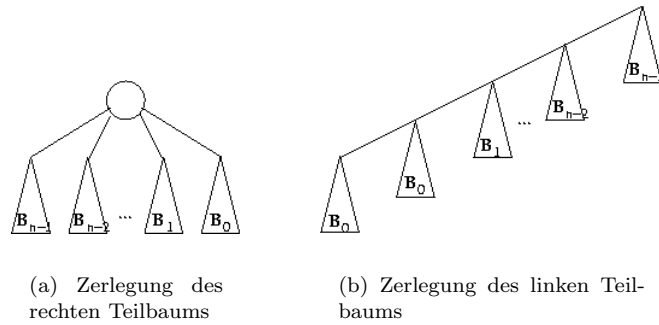
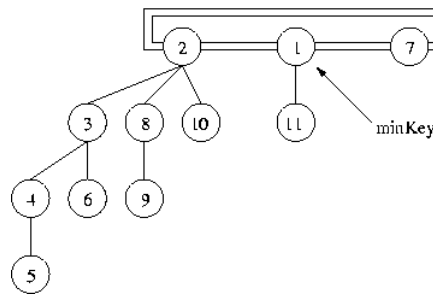
Beachte: Binomialbäume sind keine Binärbäume!
Zerlegungen von Binomialbäumen B_n : siehe Abbildung 4.2.

Proposition 16 Es sei B_n ein Binomialbaum.

1. B_n hat 2^n Knoten.
2. Die Wurzel hat n Kinder.
3. B_n hat die Höhe n .
4. B_n hat $\binom{n}{i}$ Knoten der Tiefe i .

Eine Binomial Queue mit n Objekten ist wie folgt aufgebaut:

1. Für jede Position i in der Binärdarstellung von n , an der eine 1 vorkommt, haben wir einen Binomialbaum B_i .

Abbildung 4.2: Zerlegungen des Binomialbaums B_n Abbildung 4.3: Beispiel einer Binomial Queue für $n = 11$

2. Wurzeln der Binomialbäume werden durch doppelt verkettete (zirkuläre) Listen verbunden.
3. Innerhalb jedes Binomialbaumes gilt die Heap-Eigenschaft (d.h. der Schlüssel in jedem Knoten ist höchstens so groß wie die Schlüssel in den Kindern).
4. Halte einen Zeiger auf die Binomialbaumwurzel mit dem kleinsten Schlüssel.

Beispiel Abbildung 4.3 zeigt eine Binomial Queue mit $n = 11 (\equiv 1011_2)$.

Bemerkungen

1. Ist $a_m a_{m-1} \dots a_0$ Binärdarstellung von n (mit $a_m \neq 0$), so gilt

$$n = \sum_{j=0}^m a_j 2^j = \sum_{a_j=1} 2^j = \sum_{a_j=1} \|B_j\| = \|B\|$$

2. Priority Queues sind keine guten Suchstrukturen; Priority Queues immer auch mit Suchbaum verwenden.

Komplexität der Operationen

- $\text{merge}(P, Q)$: Gehe vor wie bei binärer Addition:
 - Kommt B_i entweder in P oder in Q vor, so übernehme B_i mit Schlüsseln unverändert in neue Binomial Queue.
 - Kommt B_i sowohl in P als auch in Q vor, so baue Binomialbaum B_{i+1} mit kleinerer Wurzel der beiden B_i als neue Wurzel (entspricht Übertrag).
 - Kommt B_i durch Übertrag drei Mal vor, so wähle zwei B_i beliebig aus und verschmelze sie (d.h., es bleibt genau ein B_i übrig).

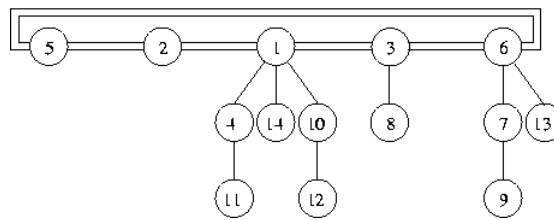


Abbildung 4.4: Beispiel eines Fibonacci-Heaps

Seien $n_1 = \|P\|$, $n_2 = \|Q\|$. Ein einzelner Merge-Vorgang kostet $O(1)$ Schritte. Dieser muss maximal so oft durchgeführt werden, wie der maximale Typ der Binomial Queue ist: $O(\log(n_1 + n_2)) = O(\log n)$.

- **insert(o, k)**: Verschmelze Binomial Queue mit neuer Binomial Queue, die nur aus B_0 mit Schlüssel k besteht: $= (\log n)$ Schritte im worst case.
- **extractMin**: Der minimale Schlüssel steht in der Wurzel eines Binomialbaums B_n . Wird ein Knoten gelöscht, so erhalten wir k Binomialbäume B_0, B_1, \dots, B_{k-1} (siehe Zerlegung 1). Fasse diese als Binomialqueue Q auf und verschmelze Q mit der verbleibenden Binomialqueue $P' = „P - B_n“$.
 $\Rightarrow O(\log(n))$ Schritte im worst case.
- **decreaseKey(k, Δ)**: Sei x ein Knoten mit dem Schlüssel k und sei $k' := \Delta$ der neue Schlüssel in x . Stelle Heap-Eigenschaft im Binomialbaum B_i , in dem sich x befindet, wieder her; durch Vertauschungen wie bei (Standard-)Heaps.
 $\Rightarrow O(k(B_i))$ Schritte.
 $\Rightarrow O(\log(n))$ Schritte im worst case.

4.4 Höhere Heapstrukturen mit Fibonacci-Heaps

(1984 von Fredman und Tarjan eingeführt)

Fibonacci-Heaps haben bessere amortisierte Komplexität als Binomial-Queues, insbesondere **decreaseKey** in $O(1)$ amortisiert.

Ein FIBONACCI-HEAP H ist eine Menge von Bäumen $\{T_1, T_2, \dots, T_m\}$ mit folgenden Eigenschaften:

1. Alle T_i erfüllen die Heap-Eigenschaft für jeden Knoten.
2. Wurzeln der Bäume werden durch doppelt verkettete Zirkulär-Liste verbunden.
3. Es wird ein Zeiger auf Wurzel mit minimalem Schlüssel gehalten.
4. Jeder Knoten in H besitzt eine Marke aus $\{0, 1\}$.
5. In jedem Knoten x wird auch die Anzahl seiner Kinder, d.h. der Grad $\deg(x)$ gespeichert.

Hilfsoperationen

1. Baumverlinkung: Verschmelze Bäume mit Wurzel y und Wurzel x zu neuem Baum unter Beachtung der Schlüsselwerte, d.h. ist $\text{key}(y) > \text{key}(x)$, dann wird der Baum mit Wurzel y linkerster Teilbaum von x — vgl. Abbildung 4.5.
2. kaskadierendes Abschneiden: Algorithmus `cascadingCut(y)`

```

1  z := Parent( y );
2  IF z != NIL
3      IF y "nicht markiert";
4          "markiere y";

```

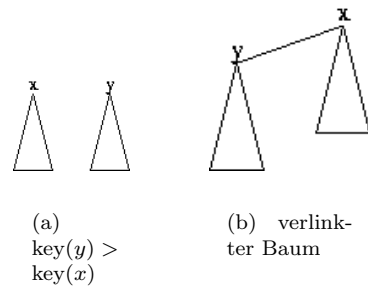


Abbildung 4.5: Baumverlinkung

```

5     ELSE
6     "entferne y aus der Kinderliste von z";
7     "mache es zum Baum in H";
8     CascadingCut( z );

```

Definition: Potential Φ eines Fibonacci-Heaps H $\Phi(H) =_{\text{def}} A(t(H) + 2 * m(H))$, wobei

- $t(H)$: Anzahl der Bäume im Heap H
- $m(H)$: Anzahl markierter Knoten in H
- A : Konstante (wird später festgelegt)

Amortisierte Komplexität der Operationen

- **insert(o, k)**: Füge den nur aus einem Knoten x mit Schlüssel k bestehenden Baum in die Wurzelliste; $\text{deg}(x) = 0$, x ist unmarkiert, aktualisiere Zeiger auf Minimum
 $\Rightarrow O(1)$ Schritte
 Außerdem: $\Delta\Phi = A((t(H) + 1 + 2 * m(H)) - (T(H) + 2 * m(H))) = A$
 $a_i = t_i + \Delta\Phi$
 $\Rightarrow O(1)$ Schritte amortisiert
- **merge**: Verbinde beide Wurzellisten und aktualisiere die Zeiger auf das Minimum
 $\Rightarrow O(1)$ Schritte
 Außerdem: $\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$
 $\Rightarrow O(1)$ Schritte amortisiert
- **extractMin**: ExtractMin ist auch Aufräumfunktion. Gehen zunächst vor wie bei **extractMin** für Binomial-Queues: Sei x der Knoten mit minimalem Schlüssel k . Gebe k zurück, entferne x und hänge alle Teilbäume der Kinder in die Wurzelliste (formal: Merge).

Jetzt: Aufräumen

Es werden immer zwei Bäume mit Wurzeln vom selben Grad verlinkt. Sei D der maximale Grad eines Knotens:

Algorithmus: Consolidate

```

1  "initialisiere ein Feld A[0 ... D] mit NIL"
2  FOR "alle Knoten in der Wurzelliste von H" DO
3    x := w
4    d := deg( x )
5    WHILE A[d] != NIL
6      y := A[d]
7      "Baumverlinkung von x und y"
8      A[d] := NIL
9      d := d + 1

```

```

10     A[d] := x                               // o.B.d.A key( x ) < key( y )
11     "aktualisiere Zeige auf Minimum mit Hilfe von A"

```

Laufzeit:

- $O(D)$ für Initialisierung (Zeile 1), Minimum-Zeiger-Aktualisierung (Zeile 11)
- $O(D + t(H))$ wegen: $D + t(H) - 1$ Knoten in Wurzelliste zur Beginn des Aufräumens und in jedem Schritt von (3) bis (9) werden zwei Wurzeln verschmolzen, d.h. maximal $D + t(H) - 1$ viele.
 $\Rightarrow O(D + t(H))$ Schritte

Außerdem:

$$\begin{aligned}
 \Delta\Phi &\leq A(D + \log t(H) + 2m(H)(t(H) + 2m(H))) \\
 &= O(D + t(H)) + A(D + \log t(H) - t(H)) \\
 &= O(D + \log t(H)) + O(t(H)) - At(H) \\
 &= O(D + \log t(H)) \quad \text{für hinreichend großes } A
 \end{aligned}$$

$\Rightarrow O(\log n)$ Schritte amortisiert, falls $D = O(\log n)$.

- **decreaseKey(k, Δ)**: (ohne Suche) Sei x Knoten mit Schlüssel k . Entferne x aus dem Baum, füge x in Wurzelliste ein (und damit den an x hängenden Teilbaum), setze $\text{key}(x) = \Delta$ und führe kaskadierendes Abschneiden an $\text{parent}(x)$ durch. Aktualisiere Zeiger auf Minimum.

$\Rightarrow O(c)$ Schritte, wobei c die Anzahl rekursiver Aufrufe von Cascading Cut ist (obere Schranke ist die Höhe von H).

Außerdem:

$$\begin{aligned}
 \Delta\Phi &= A(t(H) + c + 2(m(H) - c + 1) - (t(H) + 2m(H))) \\
 &= A(2 - c)
 \end{aligned}$$

$\Rightarrow O(c) + A(2 - c) = O(1) + O(c) - Ac = O(1)$ Schritte amortisiert (für A groß genug).

- **delete(k)**: (ohne Suche) **decreaseKey(k, $-\infty$)** + **extractMin**
 $\Rightarrow O(1) + O(\log n) = O(\log n)$ Schritte amortisiert (falls $D = O(\log n)$).

Satz 17 Für den maximalen Grad D eines Knotens in einem Fibonacci-Heap mit n Elementen gilt $D = O(\log n)$.

Damit: Laufzeitverhalten von Fibonacci-Heaps

Operation	amortisiert	worst case
insert	$O(1)$	$O(1)$
merge	$O(1)$	$O(1)$
minKey	$O(1)$	$O(1)$
extractMin	$O(\log n)$	$O(1)$
decreaseKey	$O(1)$?
delete	$O(\log n)$	$O(n)$

4.5 Höhere Heap-Strukturen: Soft Heaps

(2000 von Bernard Chazelle eingeführt)

Ziel: Operationen **insert**, **merge**, **extractMin** und **delete** in amortisiert konstanter Zeit realisieren.

Idee:

- »car pooling«: bewege mehrere Schlüssel gleichzeitig im Heap (ohne die Heap-Ordnung zu zerstören).

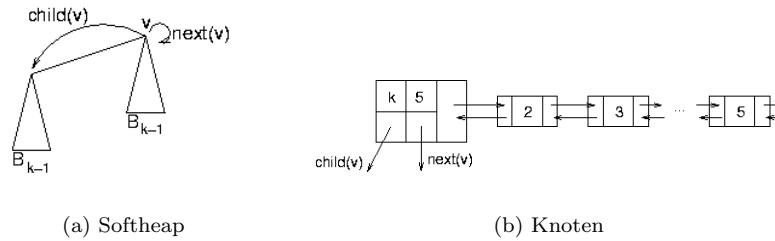


Abbildung 4.6: Softhead-Schema

- Konsequenz: es müssen »verdorbene« Schlüssel existieren, d.h. Schlüsselwerte, die nicht mehr ihrem Originalwert entsprechen. Wir müssen dafür sorgen, dass nicht zu viele verdorbene Schlüssel existieren.

Eine SOFT QUEUE ist ein Binomialbaum, in dem Knoten (und Teilbäume) gelöscht sein können. Der Binomialbaum, von dem eine Soft Queue abgeleitet worden ist, heißt REFERENZBAUM.

Der Rang eines Knotens in einer Soft Queue ist der Rang des Knotens im Referenzbaum, d.h. die Anzahl der Kinder des Knotens im Referenzbaum.

Im Folgenden halten wir die RANGINVARIANTE aufrecht: Die Anzahl der Kinder in Wurzel r einer Softqueue ist

$$\geq \left\lfloor \frac{1}{2} \text{Rang}(r) \right\rfloor$$

Die Knoten einer Soft Queue enthalten folgende Information:

- Rang des Knotens
- den maximalen Schlüssel der angeschlossenen Schlüsselliste (»gemeinsamer Schlüssel«)
- Zeiger auf nächstes Kind in der Soft Queue
- Zeiger auf nächste Soft Queue

Schema (Soft Queue = Binomialbaum) s. Abb. 4.6 ($\text{next}(v)$): Zeiger auf Wurzel der Soft Queue mit Rang $R(v) - 1$ und mit v als Wurzel, $\text{child}(v)$: Zeiger auf Wurzel der Soft Queue mit Rang $R(v) - 1$ und v ist Elternknoten der Wurzel).

Die Knoten einer Soft Queue enthalten folgende Informationen:

- Rang des Knotens
- maximaler Schlüssel der angeschlossenen Schlüsselliste
- Zeiger auf nächsten Knoten der Soft Queue

Beispiel einer Soft Queue: Abbildung 4.7

Repräsentation von Soft Queues durch Binärbäume

(dazu Abbildung 4.8) Beachte: Alle Knoten der Soft Queue kommen als Blätter vor. Außerdem:

1. In Soft Queues ist in allen Knoten die Heap-Eigenschaft bezüglich gemeinsamer Schlüssel erfüllt
2. Für $r = r(\varepsilon)$ (mit ε Fehlerrate) sind alle verdorbenen Schlüssel in Knoten enthalten, die einen Rang größer r besitzen.

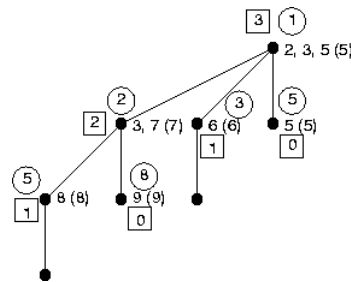


Abbildung 4.7: Beispiel einer Soft Queue (in den Kreisen: fehlende Knoten, in den Quadraten: Rang)

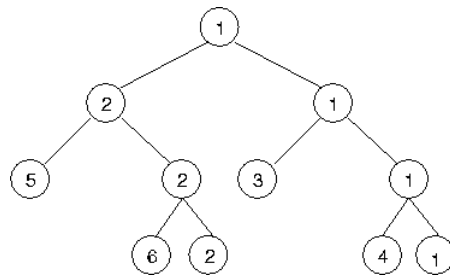


Abbildung 4.8: Soft Queue als Binärbaum

Ein SOFT HEAP ist eine Folge von Soft Queues mit paarweise verschiedenen Referenzbäumen, d.h. jeder Typ kommt höchstens ein Mal vor.

Repräsentation: Ein Soft Heap wird als doppelt verkettete Liste der Wurzeln der Soft Queues implementiert (geordnet nach Rang); für jede Soft Queue vom Typ i wird ein Zeiger auf eine Soft Queue vom Typ $j \geq i$ aufrecht erhalten, deren Wurzel den minimalen gemeinsamen Schlüssel aller dieser Soft Queues enthält.

Operationen

- **insert(o, k):** Bilde Soft Queue Q' mit Referenzbaum vom Typ 0 und Inhalt k (= gemeinsamer Schlüssel); verschmelze Q' mit bestehendem Soft Heap.
- **merge(h1, h2):** Verschmelze $h1$ und $h2$ wie bei Binomial Queues; Gehe von rechts nach links (d.h. von Soft Queues von hohem Typ zu niedrigen Typen) und aktualisiere die Minimum-Zeiger.
Beachte: wird nur einzelne Soft Queue verschmolzen, dann Aktualisierung der Minimum-Zeiger nur von Einfügeposition zurück bis Soft Queue vom Typ 0.
- **extractMin:** Der Minimum-Zeiger in der Wurzel der Soft Queue vom niedrigsten Typ zeigt auf den aktuell minimalen gemeinsamen Schlüssel im Soft Heap.
Problem: Durch andere Operationen könnte die Schlüssel-liste leer sein! Wir müssen die Liste erst auffüllen, bevor wir das Minimum extrahieren. Es sei $I(v)$ die an v hängende Schlüssel-liste.

Algorithmus: **sift(v)**

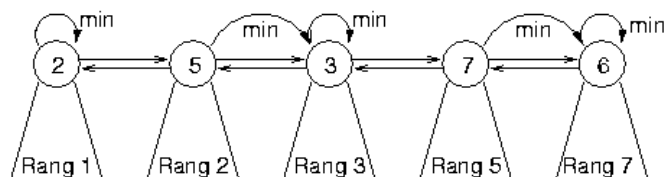


Abbildung 4.9: Soft Heap

```

1  i := 0
2  T := {}
3  IF child( v ) = NIL
4    CommonKey( v ) := infinity
5    return
6  REPEAT
7    sift( next( v ) )
8    IF CommonKey( child( v ) ) < CommonKey( next( v ) )
9      Vertausche child( v ) und next( v )
10   T := T union I( next( v ) )
11   i := i + 1
12  UNTIL Bedingung B gilt nicht
13  I( v ) := T

```

Bedingung B:

- $i \leq 1$ (höchstens zwei rekursive Aufrufe)
- $R(v) > r$ und entweder $R(v)$ ist ungerade oder $R(v) > R(\text{child}(v)) + 1$

Jetzt: Es sei v der Knoten mit minimalem gemeinsamen Schlüssel im Soft Heap. Gilt $I(v) = \emptyset$, so verfare wie folgt:

- Bestimme die Anzahl der Kinder von v in der Soft Queue (d.h. die Anzahl der `next`-Aufrufe, bis NIL erreicht wird).
- Ist die Anzahl $\geq \lfloor \frac{1}{2}R(v) \rfloor$, so führe `sift` aus, bis die Schlüsselliste nicht mehr leer ist, aktualisiere die Minimum-Zeiger und entferne Knoten mit gemeinsamem Schlüssel ∞ .
- Ist die Anzahl $< \lfloor \frac{1}{2}R(v) \rfloor$, so löse die Soft Queue mit Wurzel v auf und verschmelze alle Soft Queues, die an v hängen, mit dem Soft Heap.

Gebe ersten Schlüssel aus der Schlüsselliste zurück und entferne ihn aus der Schlüsselliste.

Beispiel (ohne Branching) Schlüssel 2, 3, 5, 1, 4, 10, 7, 6 in Soft Heap eingefügt. Die Entwicklung des Baums durch einige Aufrufe von `extractMin` wird in Abbildung 4.10 dargestellt.

Wir wollen den Anteil verdorbener Schlüssel bestimmen.

Lemma 18 Für alle Knoten v in einem Soft Heap gilt

$$\|I(v)\| \leq \max \left\{ 1, 2^{\lceil \frac{1}{2}R(v) \rceil - \frac{1}{2}r} \right\}$$

Beweis. Schlüssellisten werden nur durch `sift` vergrößert. Ohne Ausführung von `sift` gilt $\|I(v)\| = 1$ für alle v . Angenommen, `sift` wird ausgeführt. Wir zeigen die Aussage durch Induktion über den Rekursionsbaum:

IA Stößt `sift` auf ein Blatt v , so gilt $\|I(v)\| = 1$.

IS Sei der Knoten v kein Blatt. Zwei Fälle sind möglich:

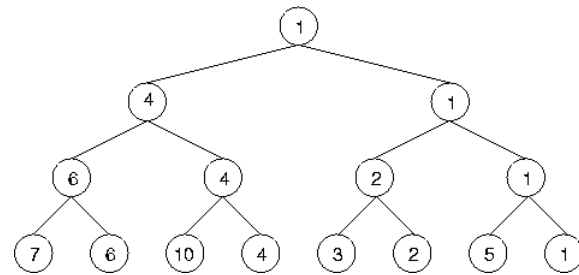
1. Bedingung B gilt nicht, d.h.

$$\|I(v)\| = \|I(\text{next}(v))\| \leq \max \left\{ 1, 2^{\lceil \frac{1}{2}R(\text{child}(v)) \rceil - \frac{1}{2}r} \right\} \leq \max \left\{ 1, 2^{\lceil \frac{1}{2}R(v) \rceil - \frac{1}{2}r} \right\}$$

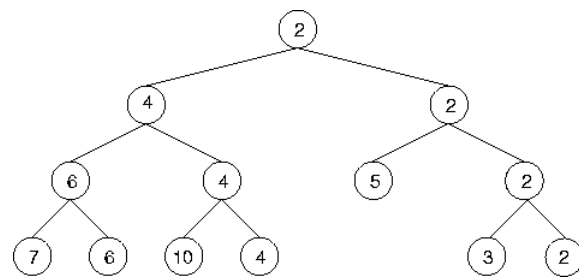
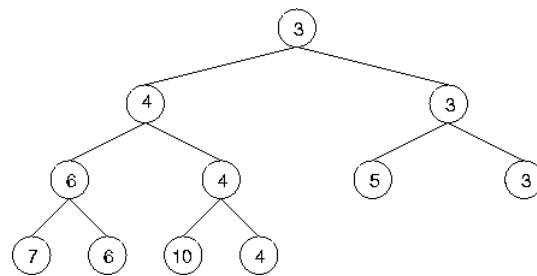
2. Bedingung B gilt, d.h. $R(v) > r$ und zwei Unterfälle sind möglich:

- (a) $R(v)$ ungerade: Es seien u, u' Knoten, auf denen `sift` rekursiv ausgeführt wird, dann gilt $R(u), R(u') < R(v)$. Damit:

$$\max\{\|I(u)\|, \|I(u')\|\} \leq \max \left\{ 1, 2^{\lceil \frac{1}{2}(R(v)-1) \rceil - \frac{1}{2}r} \right\} = 2^{\lceil \frac{1}{2}(R(v)-1) \rceil - \frac{r}{2}} = 2^{\lceil \frac{1}{2}R(v) \rceil - \frac{r}{2} - 1}$$



(a) vollständig aufgebauter Baum

(b) nach erstem `extractMin`(c) nach zweitem `extractMin`Abbildung 4.10: `extractMin` in einem Soft Heap

(b) $R(v)$ gerade und $R(\text{child}(v)) + 1 < R(v)$. Übungsaufgabe: zu zeigen:

$$\max\{\|I(u)\|, \|I(u')\|\} \leq 2^{\lceil \frac{1}{2}R(v) \rceil - \frac{r}{2} - 1}$$

Insgesamt:

$$\|I(v)\| = \|I(u) \cup I(u')\| \leq 2 \cdot 2^{\lceil \frac{1}{2}R(v) \rceil - \frac{r}{2} - 1}$$

□

Lemma 19 Für einen Binomialbaum vom Typ k gilt

$$\sum_{v \in B_k} 2^{\frac{1}{2}R(v)} \leq 2^{k+2} - 3 \cdot 2^{\frac{1}{2}k} \quad (\leq 4 \cdot 2^k)$$

Beweis. Induktion über k

IA $k = 0$: $R(v) = 0$ für einzigen Knoten v . Damit ergibt sich $1 \leq 4 - 3 = 1$.

IS $k > 0$: Die Ränge aller Knoten außer dem der Wurzel bleiben gleich, wenn B_k in zwei B_{k-1} zerlegt wird; die Wurzel hat in B_k den Rang k und in B_{k-1} den Rang $k - 1$. Damit:

$$\begin{aligned} \sum_{v \in B_k} 2^{\frac{1}{2}R(v)} &= 2 \sum_{v \in B_{k-1}} 2^{\frac{1}{2}R(v)} + 2^{\frac{1}{2}k} - 2^{\frac{1}{2}(k-1)} \\ &\leq 2 \left(2^{k+1} - 3 \cdot 2^{\frac{1}{2}(k-1)} \right) + (\sqrt{2} - 1) 2^{\frac{1}{2}(k-1)} \\ &= 2^{k+2} - \left(3 \cdot 2^{\frac{1}{2}(k+1)} - (\sqrt{2} - 1) 2^{\frac{1}{2}(k-1)} \right) \\ &= 2^{k+2} - 3 \cdot 2^{\frac{1}{2}k} \left(\sqrt{2} - \frac{1}{3\sqrt{2}} (\sqrt{2} - 1) \right) \\ &\leq 2^{k+2} - 3 \cdot 2^{\frac{1}{2}k} \end{aligned}$$

□

Satz 20 Zu jedem Zeitpunkt gibt es in einem Soft Heap mit n Schlüsseln höchstens

$$\frac{1}{2^{\frac{1}{2}r-3}} n$$

verdorbene Schlüssel.

Beweis. Für die Soft Queue Q sei $S_Q =_{\text{def}} \{v \in Q : R(v) > r\}$. Dann gilt $S_Q \subseteq S_{B_k}$ für den Referenzbaum B_k . Es gilt $\|S_{B_k}\| \leq \frac{1}{2^r} \|B_k\|$. Damit gilt für einen Soft Heap (bestehend aus Soft Queues Q_j):

$$\sum_{k_j} \|S_{Q_j}\| \leq \sum_{k_j} \|S_{B_{k_j}}\| \leq \frac{1}{2^r} \sum_{k_j} \|B_{k_j}\| = \frac{1}{2^r} n$$

Verdorbene Schlüssel sind nur ab Rang $r + 1$ möglich. Damit gibt es maximal

$$\begin{aligned} \sum_{k_j} \left(\sum_{v \in B_{k_j}} 2^{\frac{1}{2}(R(v)+1-r)} \right) &= 2^{-\frac{1}{2}(r-1)} \sum_{k_j} \sum_{v \in B_{k_j}} 2^{\frac{1}{2}R(v)} \\ &\leq 2^{-\frac{1}{2}(r-1)} 4 \frac{1}{2^r} \sum_{k_j} \|B_{k_j}\| \\ &= \frac{8}{2^{r-\frac{1}{2}r}} n = 2^{-\frac{1}{2}(r-3)} \end{aligned}$$

□

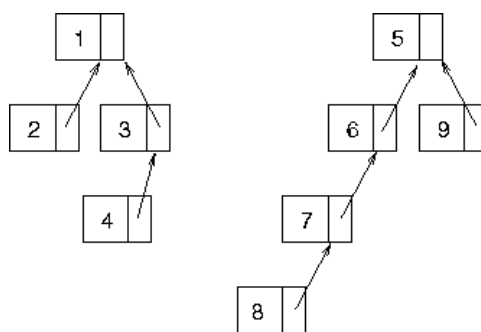


Abbildung 4.11: Menge als Baum

Komplexität der Operationen

Wir betrachten Folgen von Operationen `insert`, `merge` und `extractMin` mit mindestens n `insert`-Operationen:

- `insert(o, k)`: $O(1)$ amortisiert (wie bei Binomial Queue)
- `merge`: $O(1)$ amortisiert (ohne Beweis)
- `extractMin`: Wesentliche Operation ist `sift`. Es sei (τ_1, \dots, τ_k) die absteigende Folge der Ränge entlang eines Pfades von der Wurzel zum Blatt im Berechnungsbaum von `sift`. Wir sagen, τ_i sei »gut« genau dann, wenn τ_i ungerade ist oder $\tau_{i+1} \leq \tau_i - 2$. Dann gilt: Für $k \geq r$ ist ab r mindestens in jedem zweiten Schritt Bedingung B erfüllt.
 $\Rightarrow O(rC)$ Schritte, wobei C angibt, wie oft Bedingung B gilt.
 Es gilt: Gibt es in einer Soft Queue mit Wurzel v mindestens zwei verschiedene Schlüssel, dann werden bei Bedingung B zwei nicht leere Schlüssellisten vereinigt. Es gibt höchstens n derartige Vereinigungen, d.h. $C \leq n$. Damit: $O(r)$ Schritte amortisiert.
 \Rightarrow setze $r = 5 + \lceil 2 \log \frac{1}{\varepsilon} \rceil$. Dann: `extractMin` in $O(\log \frac{1}{\varepsilon})$ Schritten amortisiert und zu jedem Zeitpunkt sind höchstens εn verdorbene Schlüssel im Soft Heap (Satz 20).

4.6 Mengen und Union-Find-Strukturen

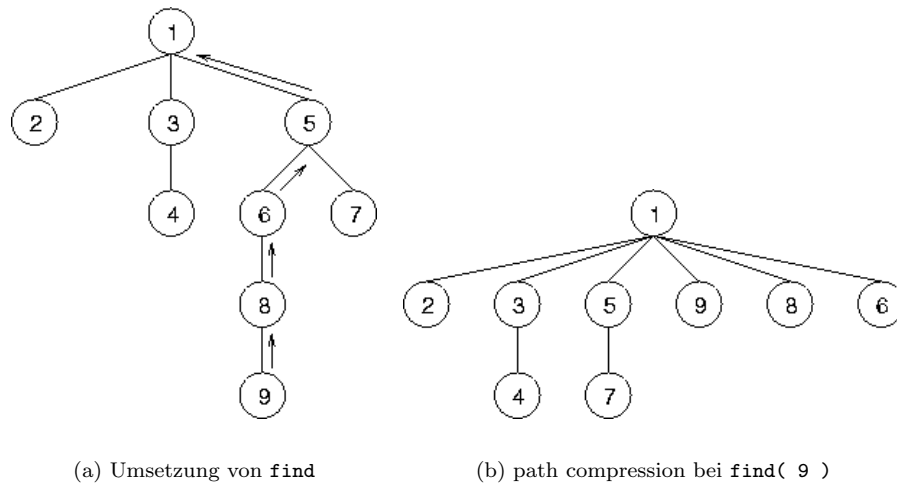
Wir interessieren uns nun für die Verwaltung von Mengen und Mengenfamilien. Mengenoperationen:

- `union(A, B)`: Ersetze A durch $A \cup B = \{x : x \in A \vee x \in B\}$
- `intersect(A, B)`: Ersetze A durch $A \cap B = \{x \in A \wedge x \in B\}$
- `subtract(A, B)`: Ersetze A durch $A \setminus B = \{x : x \in A \wedge x \notin B\}$

Bei Implementierung mit geordneten Folgen sind diese Operationen mit $O(n)$ Schritten im worst case umsetzbar.

Operationen auf Familien disjunkter Mengen

- `makeSet(k)`: Erzeuge Einermenge $\{k\}$ und gebe k als Repräsentant zurück.
- `union(k_A, k_B)`: Vereinige die Mengen A und B , die durch k_A und k_B repräsentiert werden, und gebe Repräsentanten $k_{A \cup B}$ zurück.
- `find(k)` Gib den Repräsentanten für die Menge zurück, in der sich k befindet.

Abbildung 4.12: `find` und path compression

Implementierung als Bäume

- `makeSet`: $O(1)$ Schritte
- `union(k_1 , k_2)`: hänge Bäume zusammen, z.B. wie in Abbildung 4.12 (a) $\Rightarrow O(1)$ Schritte
- `find(k)`: Folge dem Pfad bis in die Wurzel: $O(n)$ Schritte.

Besser: folgende Heuristiken verwenden:

- union-by-size: Speichere für jeden Knoten x die Größe des Teilbaumes inklusive x bei x ab; bei Vereinigung mache immer den Baum geringerer Größe zum Teilbaum des Baumes größerer Größe.
- path compression: Mache jeden Knoten, der während der `find`-Operation besucht wird, zum Kind der Wurzel.

Amortisierte Analyse von Union-Find-Folgen

Definiere folgende Funktionen:

$$t(0) =_{\text{def}} 1$$

$$t(n) =_{\text{def}} 2^{t(n-1)} \quad \text{für } n \geq 1$$

und

$$\log^*(n) =_{\text{def}} \min\{i : t(i) \geq n\}$$

d.h. \log^* ist die Umkehrfunktion von t .

Beispiel

$$t(0) = 1$$

$$t(1) = 2$$

$$t(2) = 2^2 = 4$$

$$t(3) = 2^{2^2} = 2^4 = 16$$

$$t(4) = 2^{2^{2^2}} = 2^{2^6} = 65536$$

$$t(5) = 2^{65536} \geq 10^{19728} \gg 10^{81} \text{ (vermutete Anzahl Atome im Weltall)}$$

Umgekehrt: $\log^*(t(5)) = 5$, d.h. $\log^*(n) \leq 5$ für »alle« praktisch relevanten Fälle.

Ziel: Komplexität von n **union-find**-Operationen ist $O(\log^*(n))$ amortisiert.

Wir definieren zunächst Hilfsfunktionen: Es sei T' der Baum, der durch eine Folge von **union**-Operationen (ohne **find**) entstehen würde. Sei x Knoten in t .

1. $\text{rank}(x) =_{\text{def}} h(T'(x))$ (Höhe des Teilbaums in T' mit Wurzel x)
- 2.

$$\text{class}(x) =_{\text{def}} \begin{cases} 0 & \text{falls } \text{rank}(x) = 0 \\ i & \text{falls } t(i-1) < \text{rank}(x) \leq t(i) \end{cases}$$

3. $\text{dist}(x) =_{\text{def}}$ Anzahl Kanten auf dem Pfad zur Wurzel bis zum tiefsten Vorgänger y mit $\text{class}(x) < \text{class}(y)$ oder y ist Wurzel.

Lemma 21

1. Ein Baum T einer **union-find**-Struktur besitzt mindestens $2^{h(T)}$ Knoten.
2. In einer **union-find**-Struktur gibt es maximal $\frac{n}{2^r}$ Knoten mit Rang r .

Beweis. 1. Induktion über Höhen:

IA $h(T) = 0$: Ein Baum der Höhe 0 besitzt genau einen Knoten.

IS $h(T) > 0$: T entstand durch Anhängen eines Baumes mit Höhe $h(T) - 1$, d.h. mit $\geq 2^{h(T)-1}$ Knoten (nach Induktionsvoraussetzung). Nach **union-by-size** gilt: Anzahl der Knoten in $T \geq 2^{h(T)-1} + 2^{h(T)-1} = 2^{h(T)}$.

2. Verschiedene Knoen mit gleichem Rang haben unterschiedliche Kinder. Nach (1) besitzt ein Knoten mit Rang r mindestens 2^r Nachfolger (sich selbst eingeschlossen).
 \Rightarrow max. $\frac{n}{2^r}$ Knoten mit Rang r (sonst würde es mehr als n Knoten geben). □

Wir definieren das Potential Φ wie folgt:

$$\Phi =_{\text{def}} A \sum_x \text{dist}(x) \quad \text{für geeignetes } A > 0$$

- **union**: Hintereinanderausführung von n Unions kostet $O(n)$ Schritte. Die gesamte Potentialänderung für alle **union**-Operationen ist beschränkt durch das Maximum von Φ (zur ersten Abschätzung: maximaler Rang ist $\log n$ und damit maximale Klasse $\log^* \log n = \log^*(n) - 1$):

$$\begin{aligned} \Phi &\leq A \left(\sum_{i=0}^{\log^*(n)-1} \sum_{\substack{j=t(i-1)+1 \\ \text{rank}(x)=j}}^{t(i)} \text{dist}(x) \right) \\ &\leq A \left(\sum_{i=0}^{\log^*(n)-1} \sum_{j=t(i-1)+1}^{t(i)} \frac{n}{2^j} (t(i) - t(i-1)) \right) \\ &\leq A \left(\sum_{i=0}^{\log^*(n)-1} n(t(i) - t(i-1)) \sum_{j=0}^{t(i)-t(i-1)-1} \frac{1}{2^{j+t(i-1)+1}} \right) \\ &\leq A \left(\sum_{i=0}^{\log^*(n)-1} \frac{n}{2^{t(i-1)+1}} (t(i) - t(i-1)) 2 \right) \\ &= A \left(\sum_{i=0}^{\log^*(n)-1} n \frac{t(i) - t(i-1)}{t(i)} \right) \\ &\leq An \log^*(n) \end{aligned}$$

- **find**: (ohne Suche) Es sei x Knoten mit Schlüssel k . Es sei $x = x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_m$ der Pfad von x zur Wurzel, d.h. $d_x = 1 + m$.

$\Rightarrow O(d_x)$ Schritte

Wir wollen nun $\Delta\Phi$ abschätzen. Betrachten wir dazu die Kante (x_{i-1}, x_i) .

- 1. *Fall*: $\text{class}(x_{i-1}) = \text{class}(x_i)$ und $i < m$
 \Rightarrow vor Ausführung von **find**: $\text{dist}(x_{i-1}) \geq 2$, nach Ausführung von **find**: $\text{dist}(x_{i-1}) = 1$.
 \Rightarrow Beitrag $\leq -A$ zu $\Delta\Phi$.
- 2. *Fall*: $\text{class}(x_{i-1}) < \text{class}(x_i)$ oder $i = m$
 \Rightarrow Beitrag 0 zu $\Delta\Phi$.
 aber: Es gibt höchstens $\log^*(n)$ viele solcher Kanten.

Damit:

$$\Delta\Phi \leq A(\log^*(n) - d_x)$$

$\Rightarrow O(d_x) + A(\log^*(n) - d_x) = O(\log^*(n))$ Schritte amortisiert (für A groß genug)

Kapitel 5

Graphenalgorithmen

- Ein Paar $G = (V, E)$ heißt GRAPH mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$ ($n = \|V\|$, $m = \|E\|$).
- Kanten können GERICHTET sein, beschrieben als Vektor (u, v) , oder UNGERICHTET, beschrieben als Menge $\{u, v\}$.
- Ein Graph $G = (V, E)$ heißt GERICHTET, wenn alle Kanten gerichtet sind, UNGERICHTET, wenn alle Kanten ungerichtet sind, und GEMISCHT, wenn sowohl gerichtete als auch ungerichtete Kanten vorkommen.
- Knoten $u, v \in V$ heißen ADJAZENT, falls eine Kante zwischen u und v existiert.
- Eine gerichtete Kante (u, v) ist AUSGEHENDE Kante von u und EINGEHENDE Kante für v .
- Knotengrade für $u \in V$:
 - $\deg(u) =_{\text{def}}$ Anzahl adjazenter Knoten von u
 - $\text{in-deg}(u) =_{\text{def}}$ Anzahl eingehender Kanten von u
 - $\text{out-deg}(u) =_{\text{def}}$ Anzahl ausgehender Kanten von u
- Ein WEG (der Länge k) in $G = (V, E)$ ist eine Folge (v_0, v_1, \dots, v_k) mit $\forall 0 \leq i \leq k : v_i \in V$ und $(v_{i-1}, v_i) \in E$ bzw. $\{v_{i-1}, v_i\} \in E$.
- Ein KREIS ist ein Weg (v_0, v_1, \dots, v_k) mit $v_0 = v_k$.
- Ein Weg (v_0, v_1, \dots, v_k) heißt PFAD, falls $v_i \neq v_j \forall i, j \in \{0, \dots, k\}$ mit $i \neq j$.

5.1 Kürzeste Wege

Wir betrachten nun gewichtete Graphen, d.h. jeder Kante e ist ein Gewicht $w(e)$ zugeordnet. (Das Gewicht kann z.B. als Entfernung zwischen den beteiligten Knoten interpretiert werden.) Sei $p = (e_0, \dots, e_{k-1})$ ein Weg in G , bestehend aus k Kanten. Die LÄNGE (oder das Gewicht) von p ist definiert als:

$$w(p) =_{\text{def}} \sum_{i=0}^{k-1} w(e_i)$$

Die DISTANZ zwischen zwei Knoten u und v (symbolisch $d(u, v)$) ist die minimale Länge eines Weges zwischen u und v . Konvention: $d(u, v) = +\infty$, falls es keinen Weg zwischen u und v gibt.

Beachte: Kreise mit negativem Gewicht ausschließen!

Wir betrachten zunächst das Problem, für einen gegebenen Knoten die Distanzen zu allen Knoten des Graphen zu bestimmen (single-source shortest path). Es gelte $w(e) \geq 0$ für alle $e \in E$. Annahme: G ist ungerichtet.

Idee: »greedy«-Ansatz: Wähle immer die aktuell beste Kante aus.

Algorithmus: **Dijkstra**(G, v) (1959)

- Eingabe: ungerichteter Graph G mit nicht negativen Kantengewichten; Knoten $v \in V$.
- Ausgabe: Marken $D[u]$ für jeden Knoten von G , die den Abständen von v zu u entsprechen.

```

1  D[v] := 0
2  FOR u in V except v
3    D[u] := infinity
4  Baue Priority Queue Q auf Knoten von G mit Schlüsseln D
5  WHILE Q nicht leer
6    u := extractMin( Q )
7    FOR Nachbar z von u mit z in Q
8      IF D[u] + w( u, z ) < D[z]
9        D[z] = D[u] + w( u, z )
10     setze Schlüssel von z auf D[z] in Q (decreaseKey)
11  return D[u] fuer alle Knoten u

```

Wir wollen zeigen, dass Dijkstra korrekt ist.

Invariante: Wird ein Knoten u aus Q herausgezogen, so gilt $D[u] = d(v, u)$.

Begründung: (durch Widerspruch) Es sei $C \subseteq V$ die Menge aller Knoten, die nicht mehr in Q enthalten sind, d.h. $C = V \setminus Q$. Angenommen, die Invariante stimmt nicht, d.h. es gibt ein $t \in V$ mit $D[t] > d(v, t)$, wenn t in C aufgenommen wird. Sei u der erste Knoten, auf den dies zutrifft, d.h. $D[u] > d(v, u)$. Es sei P der kürzeste Weg von v nach u (sonst wäre $d(v, u) = +\infty$). Wir betrachten den Zeitpunkt, zu dem u in C aufgenommen wird. Sei z der erste Knoten in P , der nicht zu C gehört, und sei y der Vorgängerknoten von z in P . Für y gilt: $y \in C$ und $D[y] = d(v, y)$. Außerdem: $D[z] \leq D[y] + w(y, z) = d(v, y) + w(y, z)$ (folgt aus der Kantenrelaxation, Zeilen (8) und (9)). Es folgt $D[z] = d(v, z)$, da z auf dem kürzesten Weg liegt. Dann wurde jedoch u vor z in C aufgenommen, d.h. $D[u] \leq D[z]$. Es gilt: $d(v, u) = d(v, z) + d(z, u)$, da z auf dem kürzesten Weg liegt. Damit: $D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u)$. Widerspruch zur Annahme $D[u] > d(v, u)$.

Komplexität von Dijkstra: Hängt ab von der Wahl der Implementierung der Priority Queue (der Graph wird als Adjazenzlistenstruktur repräsentiert).

- Heap
 - Zeilen (1) bis (3): $O(n)$
 - Zeile (4): $O(n)$ mit bottom-up Heap-Konstruktion
 - Zeile (5): Die WHILE-Schleife wird n Mal durchlaufen
 - Zeilen (6) bis (10): je Schleifendurchlauf
 - * $O(\log n)$ für `extractMin`
 - * $O(\deg(u) \log n)$ Schritte für die Zeilen (8) bis (10)
- ⇒ Zeilen (5) bis (10): $O(\sum_{v \in V} (1 + \deg(v)) \log n) = O((n + m) \log n)$ Schritte
- Fibonacci-Heap
 - Kantenrelaxation in $O(1)$ amortisiert über alle adjazenten Knoten, d.h. für die Zeilen (5) bis (10): $O(\sum_{v \in V} (\log n + \deg(v))) = O(n \log n + m)$ Schritte.

Einfluss negativer Gewichte auf die Korrektheit von Dijkstra: siehe Abbildung 5.1. ⇒ Dijkstra gibt $D[d] = 2$ aus, obwohl $d(a, d) = 1$.

Für den Fall negativer Gewichte: Bellman-Ford-Algorithmus (benötigt $O(nm)$ Schritte).

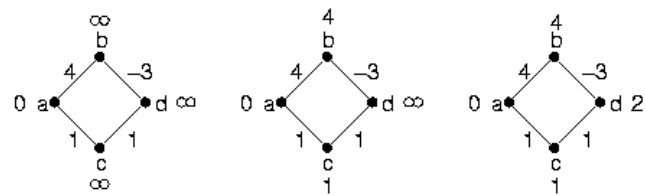


Abbildung 5.1: Dijkstra-Algorithmus mit negativen Kantengewichten