

Ausgewählte Kapitel ADS

January 14, 2016

1 Datenstrukturen für Mengen

1.1 Union-Find-Problem

Verwaltung von diskunkten Mengen

Problem

Verwalte eine Partition (Zerlegung in disjunkte Teilmengen) der Menge $\{1, \dots, n\}$ unter folgenden Operationen.
Jede Teilmenge (Block) besitzt einen eindeutigen Namen aus $\{1, \dots, n\}$.

- FIND(x): $x \in \{1, \dots, n\}$ Liefert den Namen der Teilmenge, die x enthält
- UNION(A, B, C): Vereinigt die Teilmengen mit Namen A und B zu einer Teilmenge mit dem Namen C .

Initialisierung

Wir starten mit der Partitionierung: $\{\{1\}, \dots, \{n\}\}$ mit dem Namen i für $\{i\}, 1 \leq i \leq n$

Analyse: Kosten für 1 Union (worst case)

Amortisiert: Kosten für $n - 1$ mögliche UNIONS

→ Kosten von $n - 1$ UNIONs und m FINDs

Lösungen

1. Lösung (einfach)

Verwende ein Feld $\text{name}[1..n]$ mit $\text{name}[x] = \text{Name des Blocks der } x$ enthält. $1 \leq x \leq n$

```
for i=1 to n do
    name[ i ] <- i
od
FIND(x): return name[x] :  $\mathcal{O}(n)$ 
UNION(A,B,C):  $\mathcal{O}(n)$ 
for i=1 to n do
    if name[ i ] = A OR name[ i ] = B
        then name[ i ] <- C
    fi
od
```

Gesamlaufzeit (Lemma 1):

$n - 1$ UNIONs und m FINDs kosten $\mathcal{O}(n^2 + m)$

2. Lösung (Verbesserung)

1. Find unverändert

2. Ändere den Namen der kleineren Menge in den Namen der größeren (Relabel the smaller half)

Zusätzliche Felder:

- $\text{size}[1..n]$: $\text{size}[A] = \text{Anzahl Elemente im Block } A$, initialisiert mit 1
- $L[1..n]$: $L[A] = \text{Liste aller Elemente in Block } A$, initialisiert $L[i] = \{i\}$

$\text{FIND}(x)$ bleibt gleich

$\text{UNION}(A,B)$:

```

if size [A] ≤ size [B]
then
    forall i in L[A] do
        name[i] ← B
    od
    size[B] += size[A]
    L[B] ← L[B] concatenate L[C]
else
    symmetrisch

```

Die Menge heißt jetzt A oder B

Effekt: $\text{UNION}(A,B,..)$ hat Laufzeit $\mathcal{O}(\min(|A|, |B|))$

Worst Case eines UNION dieser Folge von UNIONS: $\mathcal{O}\left(\frac{n}{2}\right) = \mathcal{O}(n)$ (kann nur einmal vorkommen)

Wie oft kann sich $\text{name}[x]$ für ein bestimmtes $x : 1 \leq x \leq n$ ändern?

Beobachtung:

- Am Anfang ist jedes Element x in einer ein-elementigen Menge
- Am Ende sind alle Elemente in einer Menge der Größe n
- Immer wenn ein Element x seinen Namen ändert befindet es sich danach in einer doppelt so großen Menge (nach dem UNION)

⇒ Jedes Element $x \in \{1, \dots, n\}$ kann maximal $\log(n)$ mal seinen Namen ändern.

Satz 1: Bei UNION-FIND mit "Relabel the smaller half" sind die Gesamtkosten einer beliebigen Folge von $n-1$ UNIONS und m Finds $\mathcal{O}(m + n * \log(n))$

Im Schnitt (amortisiert) kostet ein UNION $\log(n)$

3. Lösung

Lösung 1 und 2 haben FIND effizient gelöst, hier UNION

Jeder Block wird als Baum dargestellt. Die Knoten repräsentieren die Elemente des Blocks. In der Wurzel steht der Name des Blocks.

$\text{UNION}(A,B,E)$: Mache die Wurzel von A zum Kind der Wurzel von B und nenne die Wurzel um in E.

$\text{FIND}(x)$: Starte bei Element (Knoten) x und laufe bis zur Wurzel, dort steht der Name → $\mathcal{O}(\text{Tiefe von } x)$

Realisierung der Datenstruktur durch Felder:

$$\text{vater}[i] = \begin{cases} \text{Vater von } i \text{ in seinem Baum} \\ 0, \text{ falls } i \text{ Wurzel} \end{cases}$$

$\text{name}[i] = \text{Name des Blocks mit Wurzel } i$ (at nur Bedeutung, falls i Wurzel)

$\text{wurzel}[i] = \text{Wurzel des Blocks mit Namen } i$

Initialisierung:

```

for i=1 to n do
    vater[i] = 0
    name[i] = i
    wurzel[i] = i
od

```

FIND(x):

```

while vater[x] != 0 do
    x = vater[x]
od
return name[x]

```

UNION(A,B,C):

```

r1 = wurzel[A]
r2 = wurzel[B]
vater[r1] = r2
name[r2] = C
wurzel[C] = r2

```

Analyse:

- UNION: $\mathcal{O}(1)$ worst case
- FIND(x): Tiefe von x (max Höhe des entstehenden Baums, $n-1$ möglich)

4. Lösung (Weighted Union rule):

Vermeide große Tiefen, dafür hänge den kleineren Baum (Anzahl Knoten) an den größeren

Alternativ: Hänge den Baum mit kleinerer Höhe an den tieferen.

Realisierung: Zusätzliches Feld

$\text{size}[i]$ = Anzahl Knoten um Unterbaum mit Wurzel i

Initialisierung:

FIND(x) (wie bei 3):

```
for i=1 to n do
    vater[i] = 0      while vater[x] != 0 do
        name[i] = i          x = vater[x]
        wurzel[i] = i      od
        size[i] = 1      return name[x]
od
```

Laufzeit $\mathcal{O}(\log(n))$:

Sei für jeden Knoten x die $\text{höhe}(x)$ die Höhe von x in seinem Baum (maximale Pfad zu Blatt), Blatt=0
 $\text{size}(x)$: Anzahl der Knoten im Unterbaum mit Wurzel x (Gewicht)

Lemma: Bei weighted Union rule gilt stets, dass $\text{size}(x) \geq 2^{\text{höhe}(x)}$ für alle Knoten x.

Beweis: Induktion über $\text{höhe}(x)$:

Voraussetzung:

$\text{höhe}(x) = 0$: x ist Blatt $\rightarrow \text{size}(x) = 1 = 2^0$

Anfang:

$\text{size}(y) \geq 2^{\text{höhe}(y)}$

Schritt:

Sei $\text{höhe}(x) > 0$

Sei y ein Kind von x mit $\text{höhe}(x)-1$

Betrachte die UNION Operation bei der x zum Vater von y wurde.

Seien $\overline{\text{size}}(x)$ und $\overline{\text{size}}(y)$ die Gewichte vor der UNION Operation, dann gilt:

1) $\text{size}(y) = \overline{\text{size}}(y)$, da sich das Gewicht nur für Wurzeln ändern kann

2) $\overline{\text{size}}(x) \geq \text{size}(y)$ durch weighted union rule

3) Nach der Operation: $\text{size}(x) \geq \overline{\text{size}}(x) + \overline{\text{size}}(y)$

$\geq 2 * \overline{\text{size}}(y)$ wegen 2.

$\geq 2 * \overline{\text{size}}(y)$ wegen 1.

$\geq 2 * 2^{\text{höhe}(y)}$ nach IA

$= 2^{\text{höhe}(y)+1} = 2^{\text{höhe}(x)}$

Da Anzahl der Knoten $n \Rightarrow \text{size}(x) \leq n$ gilt:

$\Rightarrow n \geq \text{size}(x) \geq 2^{\text{höhe}(x)}$ für alle x

$\text{höhe}(x) \leq \log(n)$

Satz: Bei UNION-FIND mit weighted UNION ist die Laufzeit einer beliebigen Folge $n-1$ Unions und m Finds $\mathcal{O}(n + \log(n))$

Beweis: 1. UNION $\mathcal{O}(1)$ worst-case, 2. Find $\mathcal{O}(\log(n))$ worst case (Lemma)

5. Lösung (Verbesserung von FIND):

Pfad-Komprimierung (path compression)

Ein FIND(x) durchläuft den Pfad von x zur Wurzel.

$x = x_0, \dots, x_l = \text{Wurzel}$

Idee: Hänge x_0, \dots, x_{l-1} direkt an die Wurzel an.

Erhöht die Kosten dieses Finds um einen konstanten Faktor.

Algorithmus:

FIND(x)

```
r <- x;
while vater[r] != 0 do
    r <- Vater[r]
od
while x != r do
    y <- vater[x]
```

UNION(A,B,C):

```
r1 = wurzel[A]
r2 = wurzel[B]
if size[r1] ≤ size[r2] then
    vater[r1] = r2
    name[r2] = C
    wurzel[C] = r2
    size[r2] += size[r1]
else
    symmetrisch
```

```

vater [ x ] <- r
x <= y
od

```

Ganz klar: $\mathcal{O}(\log(n))$ worst case

Satz (Tarjan): Bei UNION-FIND mit weighted UNION und path compression hat eine beliebige Folge von n-1 Unions und m Finds mit $m \geq n$, die Gesamtkosten $\mathcal{O}(m * \alpha(m, n))$, wobei $\alpha(m, n) = \min\{z \in \mathbb{N} | A(z, \frac{4m}{n}) > \log n\}$

mit A einer Variante der Ackermannfunktion.

α ist eine Art Inverse der Ackermannfunktion \Rightarrow Ist extrem langsam wachsend.

Definition von A:

$$A : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$$

$$A(i, 0) = 0 \text{ für alle } i \in \mathbb{N}_0$$

$$A(0, x) = 2x \text{ für alle } x \geq 1$$

$$A(i, 1) = 2$$

$$A(i, x) = A(i - 1, A(i, x - 1)) \text{ für } i \geq 1, x \geq 2$$

$$\begin{array}{cccccc} 0 & 2 & 4 & 6 & 8 & 10 \\ 0 & 2 & 4 & 8 & 16 & 32 \end{array}$$

$$\begin{array}{cccccc} A(i, x) \text{ als Matrix } i \times x: & 0 & 2 & 4 & 16 & 65536 & 2^{65536} \\ & 0 & 2 & 4 & 65536 & 2 \uparrow\uparrow 65536 & . \\ & 0 & 2 & . & . & . & . \end{array}$$

$$1. \text{ Zeile: } A(0, x) = 2x; 2. \text{ Zeile: } A(1, x) = 2^x; 3. \text{ Zeile: } A(2, x) = 2^{2^x}$$

Anmerkung: Pfeilschreibweise=(Knuth Up-Arrow)

Beweis des Satzes:

Situation: n Elemente {1,...,n}, beliebige Folge von n-1 Unions und m Finds: $U_1, F_1, F_2, U_2, \dots$

Am Ende: 1 Baum T' (n-1 weighted Unions)

Konzeptuell kann T' anders erhalten werden: Führe zunächst alle Unions aus \rightarrow Baum T. Dann führe m partielle Finds auf T aus (PF_1, \dots, PF_m), die genau den selben Pfad wir F_i durchlaufen, bis zu ihrer ursprünglichen Wurzel vor den Unions.

Wir schätzen nun die Gesamtkosten dieser Folge (insbesondere der m PF's) ab.

Frage: Wieviele Vater-Verweise (Kanten) werden insgesamt durchlaufen?

Sei F=Multi-Menge aller durch die PF's durchlaufenden Kanten (mit Mehrfachen)

Zu zeigen: $|F| = \mathcal{O}(m * \alpha(m, n))$

Idee:

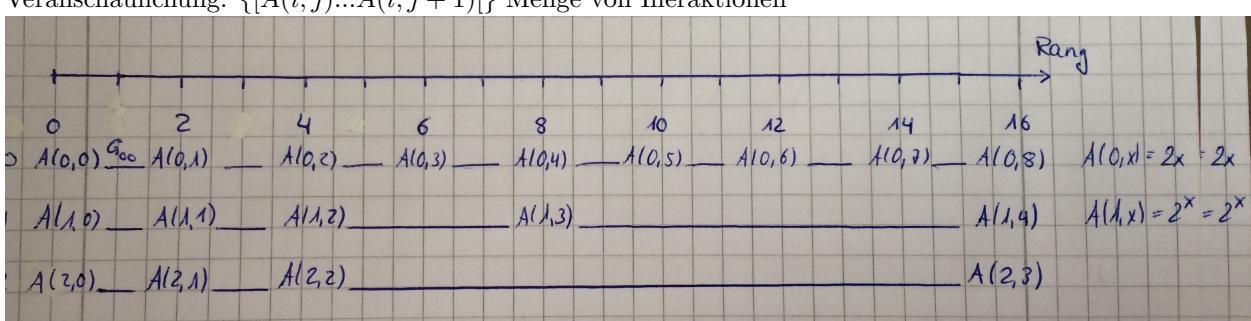
- Teile F in Gruppen nach Rang der Endpunkte der Kanten, wobei $\text{Rang}(x)=\text{Höhe}(x)$ im Baum T' (nicht T)
- Schätzt die Gruppen einzeln ab.

Zunächst Einteilung der Knoten in die Gruppen nach Rang (nicht disjunkt).

Sei $z \in \mathbb{N}_0$, Für $0 \leq i \leq z, j \geq 0$ sei:

$$G_{i,j} = \{Knoten x | A(i, j) \leq \text{Rang}(x) < A(i, j + 1)\}$$

Veranschaulichung: $\{[A(i, j) \dots A(i, j + 1)]\}$ Menge von Interaktionen



Beispiel: $\text{Rang}(x)=7$, $\text{Rang}(y)=13$

$\Rightarrow x \in G_{0,3}, G_{1,2}, G_{2,2}, \dots$

$\Rightarrow y \in G_{0,6}, G_{1,3}, G_{2,2}, \dots$

Eine Einteilung der Multi-Menge F

Für $0 \leq k \leq z$: $N_k = \{(x, y) \in F | k = \min\{i \geq 0 | \exists j \text{ mit } x, y \in G_{i,j}\}\}$

und $N_{z+1} := F \setminus \bigcup_{0 \leq i \leq z} N_i$

Schließlich definieren wir für $0 \leq k \leq z+1 : L_k = \{(x, y) \in N_k | (x, y) \text{ ist letzte (oberste) Kante auf PF-Pfad}\}$

- a) $|L_k| \leq m$ für $0 \leq k \leq z+1$
- b) $|N_0 \setminus L_0| \leq n$
- c) $|N_k \setminus L_k| \leq \frac{5}{8}n$ für $1 \leq k \leq z$
- d) $|N_{z+1} \setminus L_{z+1}| \leq n * a(z, n)$ mit $a(z, n) = \min\{i \geq 0 | A(z, i) > \log(n)\}$

Beweis:

a) Für jedes PF gibt es höchstens 1 Kante in L_k . Die Behauptung folgt daraus, dass es insgesamt nur m PFs gibt.

b) Sei $(x, y) \in N_0 \setminus L_0$, dann gilt $\exists j \geq 0$ mit $x, y \in G_{0,j}$, das heißt $A(0, j) \leq Rang(x) < Rang(y) < A(0, j+1)$
 $\Rightarrow Rang(x) = 2j, Rang(y) = 2j + 1$

$(x, y) \notin L_0 \Rightarrow$ nicht die letzte Kante in diesem PF: Betrachte PF von (x,y), dann existiert eine Kante $(s, t) \in L_0$ auf diesem PF-Pfad

Situation:

$$Rang(x) = 2j, Rang(y) = 2j + 1$$

$Rang(s) \geq Rang(y)$ da letzter Pfad vor dem nicht letzten sein muss

$Rang(t) > Rang(s)$ da Pfad von s nach t

$$\Rightarrow Rang(t) \geq 2j + 2$$

Nach dem PF: x hat neuen Vater (möglicherweise t) u mit $Rang(u) \geq Rang(t) \geq 2k + 2$

$$\Rightarrow \text{Rangdifferenz zwischen x und dem neuen Vater u} \geq 2$$

\Rightarrow Spätere PFs können keine Kante (x,..) mehr zu N_0 hinzufügen

\Rightarrow Für jeden Knoten wird maximal eine ausgehende Kante (Vaterverweis) in $N_0 \setminus L_0$ gezählt werden.

$$\Rightarrow |N_0 \setminus L_0| \leq n$$

Beweis c), d):

Idee: Schätze Beitrag eines Knotens $x \in G_{k,j}$ zu $N_k \setminus L_k$ d.h. alle Kanten, die von x ausgehen und in $N_k \setminus L_k$ gezählt werden.

Sei $k \geq 1$ und $x \in G_{k,j}$ beliebig, d.h. $\exists j$ mit $A(k, j) \leq Rang(x) < A(k, j+1)$

und y_1, \dots, y_q alle Endknoten mit $(x, y_i) \in N_k \setminus L_k$. Ziel: q nach oben abschätzen.

$$\Rightarrow Rang(y_1) \leq \dots \leq Rang(y_q) < A(k, j+1)$$

Beobachtungen:

1) $j \geq 2$ weil sonst k=0 die minimale Zeile definiert, sodass (x, y_i) im selben Intervall (die ersten 3 Spalten sind immer gleich gefüllt mit 0,2,4). Hier: $k \geq 1$

2) $(x, y_i) \notin L_k$ für $1 \leq i \leq q \Rightarrow \exists (s_i, t_i) \in N_k$ auf PF-Pfad von (x, y_i) oberhalb von (x, y_i) Nach der Pfadkopplermierung ist Vater von $x = y_i + 1$. Außerdem ist y_{i+1} Vorfahr von t_i dabei ist $t_i = y_{i+1}$ möglich.

Es gilt stets $Rang(x) < Rang(y_i) \leq Rang(s_i) < Rang(t_i) \leq Rang(y_{i+1})$

Definition von N_k k minimal $\Rightarrow (x, y_i), (s_i, t_i) \notin N_{k-1}$

$$\Rightarrow \exists j \text{ mit } Rang(s_i) < A(k-1, j) \leq Rang(t_i)$$

Daher gilt $Rang(y_i) < A(k-1, j) \leq Rang(y_{i+1})$ für $1 \leq i \leq q-1$

Anwendung auf die gesamte Folge y_1, \dots, y_q (d.h. q-1 mal):

$$Rang(y_1) < A(k-1, j_1) \leq Rang(y_2) < A(k-1, j_2) \leq Rang(y_3) < \dots \leq Rang(y_{q-1}) < A(k-1, j_{q-1}) < Rang(y_q)$$

Beobachtung: $j_{i+1} \geq j_i \Rightarrow \exists j_1 \text{ mit } Rang(y_1) < A(k-1, j_1) \leq A(k-1, j_1 + q-1) \leq Rang(y_q)$

I. $\exists j \geq 2 : Rang(y_q) \geq A(k-1, j + q - 1)$

Beweis Teil c):

$k \geq 1, x \in G_{k,j}, (x, y_i) \in N_k \Rightarrow y_1, \dots, y_q \in G_{k,j}, j \geq 2$

$$\Rightarrow A(k, j) \leq Rang(y_1) \leq \dots \leq Rang(y_q) < A(k, j+1)$$

II. $Rang(y_q) < A(k, j+1)$

Nach I und II:

$$\exists j : A(k-1, j + q - 1) \leq A(k, j+1) = A(k-1, A(k, j))$$

\Rightarrow (Monotonie von A in Zeilen) $j + q - 1 < A(k, j)$

$$\Rightarrow (j \geq 2)q < A(k, j)$$

Wir haben gezeigt: Für jeden Knoten $x \in G_{k,j}, k \geq 1, j \geq 2$ gibt es höchstens A(k,j) Kanten $(x, y) \in N_k \setminus L_k$

$$\Rightarrow |N_k \setminus L_k| \leq \sum_{j \geq 2} |G_{k,j}| * A(k, j) \text{ mit } 1 \leq k \leq z$$

Behauptung: $|G_{k,j}| \leq \frac{2n}{2^{A(k,j)}}$ extrem fallend, n Knoten im Baum.

Daraus folgt: $|N_k \setminus L_k| \leq \sum_{j \geq 2} \frac{2n * A(k, j)}{2^{A(k, j)}}$ wobei $A(k, j) \leq 2^j$, da $k \geq 1$ zweite Zeile.

$\leq 2n \sum_{j \geq 2} \frac{2^j}{2^{2^j}} = 2n \sum_{j \geq 2} \frac{1}{2^{2^j-j}} = 2n(\frac{1}{4} + \frac{1}{32} + \frac{1}{2^{12}} + \dots)$ wobei $(\frac{1}{32} + \dots)$ mit $\frac{1}{16}$ abgeschätzt wird.
 $\Rightarrow = \frac{5}{8}n$

Beweis der Behauptung $|G_{k,j}| \leq \frac{2n}{2^{A(k,j)}}$

Sei l beliebig mit $A(k,j) \leq l < A(k,j+1)$

Zähle zuerst alle Knoten mit $Rang(x) = l$. Dafür sei $G_{k,j,l} = \{x \in G_{k,j} \mid Rang(x) = l\}$

Es gilt:

1. Jeder Knoten x mit $Rang(x) = l$ hat mindestens 2^l Nachkommen (alle Knoten im Unterbaum mit Wurzel x), nach Lemma weighted Union

2. Für $x \neq y$ mit $Rang(x) = Rang(y)$ sind die Nachkommensmengen disjunkt. 1+2: $|G_{k,j,l}| \leq \frac{n}{2^l}$ mit n Gesamtanzahl der Knoten

$$\begin{aligned} \Rightarrow |G_{k,j}| &= \sum_{l=A(k,j)}^{A(k,j+1)-1} |G_{k,j,l}| \\ &\leq \sum_{l=A(k,j)}^{\inf} \frac{n}{2^l} = n \sum_{l=A(k,j)}^{\inf} \frac{1}{2^l} \\ &= n \left(\frac{1}{2^{A(k,j)}} + \frac{1}{2} \frac{1}{A(k,j)} + \frac{1}{4} \frac{1}{A(k,j)} + \dots \right) \\ &= n * \frac{2}{2^{A(k,j)}} \end{aligned}$$

Beweis Teil d:

$k = z + 1$, weighted Union $\Rightarrow Rang(y_q) \leq \log(n)$

Nach I. für $k = z + 1$:

$A(z, j + q - 1) \leq Rang(y_q) \leq \log(n)$

$\Rightarrow j + q - 1 < \alpha(z, n)$, da $\alpha(z, n)$ minimal mit $A(z, \alpha(z, n)) > \log(n)$

$\Rightarrow q < \alpha(z, n)$ mit $j \leq 2$

Also gibt es für jeden Knoten x höchstens $\alpha(z, n)$ Kanten $(x, \dots) \in N_{z+1} \setminus L_{z+1}$ mit $n = \text{Anzahl aller Knoten}$

$\Rightarrow |N_{z+1} \setminus L_{z+1}| \leq n * \alpha(z, n)$

Beweis des Satzes (Tarjan):

Jede beliebige Folge von $n-1$ Unions und $m \geq n$ Finds hat die Gesamtaufzeit von $\mathcal{O}(m * \alpha(m, n))$

Lemma (Kosten aller Finds): Für jedes $z \geq 0$ gilt:

$$\begin{aligned} |F| &= \sum_{k=0}^{z+1} |L_k| + \sum_{k=1}^{z+1} |N_k \setminus L_k| = \sum_{k=0}^{z+1} |L_k| + |N_0 \setminus L_0| + \sum_{k=1}^z |N_k \setminus L_k| + |N_{z+1} \setminus L_{z+1}| \\ &\leq (z+2) * m + n + \frac{5}{8}n * z * + n * \alpha(z, n) \end{aligned}$$

Betrachte $z = \alpha(m, n)$, $n \leq m$ (jedes z liefert eine obere Schranke)

Dann gilt:

$$|F| \leq \mathcal{O}(m * \alpha(m, n)) + n + \mathcal{O}(n * \alpha(m, n)) + n * \alpha(z, n)$$

$$|F| \leq \mathcal{O}(m * \alpha(m, n) + n * \alpha(z, n))$$

$$\Rightarrow \alpha(\alpha(m, n), n) \leq 4 \frac{m}{n}$$

$$\Rightarrow \leq n * \frac{4m}{n} = \mathcal{O}(m)$$

Insgesamt: Kosten aller Find-Operationen sind $|F| = \mathcal{O}(m\alpha(m, n) + m) = \mathcal{O}(m\alpha(m, n))$

Kosten aller Unions: $\mathcal{O}(n) = \mathcal{O}(m)$

Bemerkung:

1) In der Praxis sehr gute Laufzeiten (sehr einfache Algorithmen und Datenstrukturen, $\alpha(m, n) < 4$ für alle in der Praxis vorkommenden Werte von m, n).

2) Die Schranke $m\alpha(m, n)$ ist scharf, d.h. das Union-Find-Problem hat tatsächlich diese Komplexität. Es gibt Beweise für untere Schranke $\Omega(m * \alpha(m, n))$

3) Optimal...yeay.

4) Varianten: Split-Find (für Intervalle),

Union-Split-Find: Split(x) markiere x , Union(x) entfernt Markierung, Find(x) nächste Markierung rechts von x

1.2 Wörterbücher

Wir kennen balancierte Suchbäume.

Problem: Teilmenge $S \subseteq U$ mit U Universum, eventuell linear geordnet.

Speichere S (Schlüssel) in einer Datenstruktur D mit:

D.insert(x) $x \in U$, $S \leftarrow S \cup \{x\}$

D.delete(x) $x \in U$, $S \leftarrow S \setminus \{x\}$

D.lookup(x) $x \in U$, testet ob $x \in S$

Es soll zusätzlich zum Schlüssel Zusatzinformation gespeichert werden.

Wir behandeln 2 Datenstrukturen: Randomisierte Suchbäume, Perfektes Hashing

1.2.1 Randomisierter Suchbaum

Entwickelt von Seidel/Aragon

Idee: Verwende einen Zufallsprozess zur Balancierung von binären Suchbäumen.

Vorteile: Sehr einfache Implementierung, geringer Aufwand zur Verwaltung der Balance, effizient

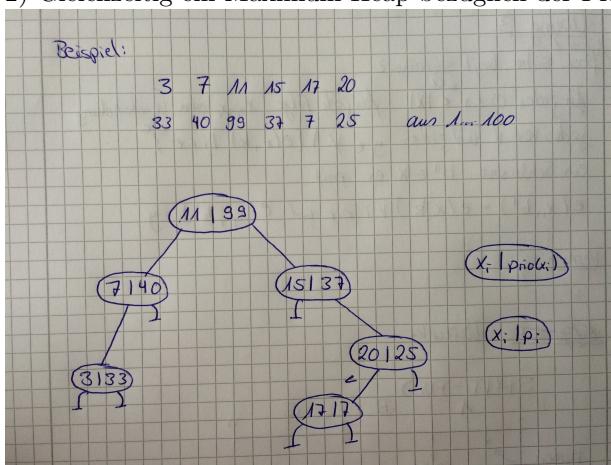
Definition RST(Randomized Search Tree):

Sei $S = \{x_1, \dots, x_n\}$ eine Menge von n Schlüsseln aus einem linear geordneten Universum U .

Jedem $x_i \in S$ wird zusätzlich eine Zufallszahl $prio(x_i)$, seine Priorität zugeordnet. Diese Zufallszahlen sind gleichverteilte reelle Zahlen aus $[0,1]$ oder praktisch ganze Zahlen aus $[0, \dots, 2^{32} - 1]$.

Ein RST für S ist:

- 1) Ein Knotenorientierter binärer Suchbaum für die Paare $(x_i, prio(x_i))$
- 2) Gleichzeitig ein Maximum-Heap bezüglich der Prioritäten $prio(x_i)$



Die Prioritäten auf einem Pfad von der Wurzel zu einem Blatt sind monoton fallend. Die maximale Priorität ist in der Wurzel.

- 1) Sie x_i der Schlüssel mit maximaler Priorität, Erzeuge den Wurzelknoten v mit dem Inhalt (x_i, p_i) mit p_i maximal.
- 2) $v.left \leftarrow (\{x_j | x_j < x_i\})$
- 3) $v.right \leftarrow (\{x_k | x_k > x_i\})$

Beobachtung: Degenerierter Baum ist unwahrscheinlich, da er nur auftritt, wenn in der Liste der maximale Schlüssel immer die maximale Priorität hat.

Erwartete Höhe (Kosten der Operationen) ist $\mathcal{O}(\log(n))$.

Alternative Konstruktion:

Füge alle Schlüssel in absteigender Reihenfolge bzgl. der Priorität in einen normalen Knoten-orientierten Baum ein. Dabei fügt das Insert den entsprechenden Knoten an die richtige Position bzgl. des Schlüssels ein. Der Baum wird durch eine zufällige (weil Prioritäten zufällig) Folge von Inserts aufgebaut.

Operationen auf einem RST:

- 1) Lookup(x): Normale Suche im binären Suchbaum. Falls $x \in S$ gilt: $\mathcal{O}(Tiefe(x))$, also maximal $\mathcal{O}(Hoehe(T))$
- 2) Insert(x): Bestimme eine zufällige Priorität $prio(x) \in [0, 1]$. Füge einen neuen Knoten (Blatt) v mit Inhalt $(x, prio(x))$ gemäß dem Schlüssel x durch ein normales Insert in den RST ein. Im Allgemeinen wird hier die Heapeigenschaft bzgl. der Prioritäten verletzt. Rotiere deswegen v nach oben, bis $prio(vater(v)) \geq prio(v)$ oder bis $v = \text{Wurzel}$

Beim Rotieren gibt es zwei Fälle:

1. v ist rechtes Kind $\rightarrow \text{rotate_left}(vater(v))$
2. v ist linkes Kind $\rightarrow \text{rotate_right}(vater(v))$

Dabei sind es meistens nur wenige auszuführende Rotationen. Der Fall, dass bis zur Wurzel rotiert werden muss ($prio(v)$ ist neues Maximum) ist selten.

- 3) Delete(x): Lookup(x) liefert Knoten v mit Schlüssel x . Rotiere v nach unten, bis er ein Blatt ist. Lösche diesen dann.

Runterschieben des zu löschenen Knotens:

```

//Sei pl die Prioritaet des linken Kindes , pr die des rechten .
while v ist kein Blatt do
    if pl > pr und v.right == null
        rotate_right(v)
    else
        rotate_left(v)
fi
od

```

4) Split(y) teilt den Unterbaum von y. $s_1 = \{x \in S | x \leq y\}, s_2 = \{x \in S | x > y\}$

5) Join(T_1, T_2): Bildet aus den beiden uebergebenen RST einen neuen. (T_1, T_2 sind RST von s_1, s_2)

Bedingung: $\max(s_1) < \min(s_2)$

Kontruiere RST mit $\max(s_1) < x < \min(s_2)$ mit x als Wurzelement mit unendlicher Prioritaet. Entferne x mit delete(x).

Laufzeitanalyse:

Wir analysieren die erwarteten Kosten einer Delete-Operation in einem RST mit n Knoten. D.h. fuer Schlüssel x_1, \dots, x_n , die durch Insert eingefügt wurden. Beobachtung: Insert ist das inverse Delete

Sei T ein RST für die Menge $S = \{x_1, \dots, x_n\}$ mit $x_1 < \dots < x_n$ entstanden durch Folge von Inserts.

Betrachte Operation Delete(x_k) mit $1 \leq k \leq n$

Allgemeine Situation: P_k : Suchpfad nach x_k , L_k : Rechtes Rückgrat vom linken Unterbaum von x_k , R_k : linkes Rückgrat vom rechten Unterbaum von x_k .

Kosten: $\mathcal{O}(|P_k| + |L_k| + |R_k|)$ mit P_k Lookup, L_k und R_k das Rotieren.

Lemma 1: Sei $S = \{x_1, \dots, x_n\}$ mit $x_i < x_{i+1}$ für $i = 1, \dots, n-1$ und $\{\text{prio}(x_i) | i = 1, \dots, n\}$ eine Menge von gleichverteilten reelen Zufallszahlen aus $[0,1]$ abgespeichert in einem RST (Betrachte den Knoten v, der einen beliebigen Schlüssel x_k enthält). Dann gilt:

a) $E(|P_k|) = H_k + H_{n-k+1} - 1$ H: Harmonische Zahl.

b) $E(|L_k|) = 1 - \frac{1}{k}$

c) $E(|R_k|) = a - \frac{1}{n-k+1}$

Wobei $H_k = \sum_{i=1}^k \frac{1}{i}$

1) $H_k \leq 1 + \ln(k)$

2) $\sum_{i=0}^{k-1} H_i = k * (H_k - 1)$

Beweis Lemma 1:

Betrachte eine Permutation $\Pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$, die S nach Prioritäten absteigend sortiert, d.h. $\text{prio}(\text{x}_{\Pi(1)}) > \text{prio}(\text{x}_{\Pi(2)}) \dots$. Dann gilt (Beobachtung):

1) Jede der Permutationen Π ist gleich wahrscheinlich (da Prioritäten gleichverteilt).

2) Man erhält den RST (denselben binären Baum) durch normales Einfügen von $x_{\Pi(1)}, \dots, x_{\Pi(n)}$ in einen normalen binären Suchbaum.

3) Dann wächst der Baum nur an den Blättern.

Durch die zufälligen Prioritäten ist der RST immer ein zufälliger Baum.

Teil a): Suchpfad P_k

Seien P'_k und P''_k eine Zerlegung von P_k : $P'_k = \{v \in P_k | \text{key}(v) \leq x_k\}, P''_k = \{v \in P_k | \text{key}(v) \geq x_k\}$

$w \in P'_k$ war zum Zeitpunkt seiner Einfügung der kleinste aller Knoten mit Schlüssel $\geq x_k$, $v \in P_k$ der größte $\leq x_k$

Wir zeigen $E(|P'_k|) = H_k$, sowie $E(|P''_k|) = H_{n-k+1}$, daraus folgt a).

Zur Abschätzung von $E(|P'_k|)$ definieren wir ein Spiel:

Spiel A: Ziehe zufällige Schlüssel aus $\{x_1, \dots, x_k\}$ und zähle wie oft der Schlüssel maximal ist. Das Spiel endet, sobald x_k gezogen wird.

A^k = erwartetes Ergebnis von Spiel A für den Schlüssel $x_k = E(|P + k'|)$

Induktion: $A^i = H_i$ für $i < k$ (Erwartungswert: $E = \sum_X (\text{prob}(x) * \text{value}(x))$

$A^k = \frac{1}{k} * 1 + \sum_{i=1}^{k-1} \frac{1}{k} * (1 - A^{k-i}) = \frac{1}{k} \sum_{i=1}^k (1 + A^{k-i}) = \frac{1}{k} (k + \sum_{i=0}^{k-1} H_i) = 1 + \frac{1}{k} (k(H_k - 1)) = H_k$

$\frac{1}{k} * 1$: Im ersten Zug x_k , $\sum_{i=1}^{k-1} \frac{1}{k} * (1 - A^{k-i})$ im 1. Zug nicht x_k sondern x_i aus $\{x_1, \dots, x_{k-1}\}$

Spiel B: Kandidaten $\{x_k, \dots, x_n\}$. Ziehe zufällige Elemente und zähle wie oft ein neues Minimum gezogen wird.

B^k = Erwartungswert dieser Zahl

Behauptung: $B^k = H_{n-k+1}$, Beweis symmetrisch zu A (Übung)

\Rightarrow Teil a) des Lemmas: $E(|P_k|) = H_k + H_{n-k+1} - 1$, weil x_k doppelt gezählt.

Wie in Teil a) können wir annehmen, dass der RST ein normaler binärer Baum mit zufälliger Insert-Reihenfolge

$x_{\Pi}(1), \dots, x_{\Pi}(n)$ ist.

L_k : Ziehe zufällige Elemente sobald x_k gezogen ist (Trigger). Zähle wie oft ein Element $> x_k$ gezogen wird, das maximal ist.

R_k : symmetrisch mit Element $< x_k$ minimal.

Spiel C: k Kandidaten $\{x_1, \dots, x_k\}$

$$C^k := E(|L_k|) = \frac{1}{k} * A^{k-1} + \sum_{i=1}^{k-1} (\frac{1}{k} C^{k-i})$$

$$C^k = \frac{1}{k} (H_{k-1} + \sum_{i=0}^{k-1} C^i)$$

Trick: schätze $\Delta_j := C^{j+1} - C^j$ ab $\Rightarrow \sum_{i=0}^{k-1} C^i = \sum_{i=1}^{k-1} (\Delta_j)$

Betrachte: $(j+1) * C^{j+1} - j * C^j = C^j + H_j - H_{j-1}$

$$\Rightarrow (j+1) * (C^{j+1} - C^j) = H_j - H_{j-1}$$

$$\Rightarrow \frac{1}{j*(j+1)} = C^{j+1} - C^j$$

$$\Rightarrow \Delta_j = \frac{1}{j} - \frac{1}{j+1}$$

$$C^k = \sum_{j=1}^{k-1} \Delta_j = \sum_{j=1}^{k-1} (\frac{1}{j} - \frac{1}{j+1}) = 1 - \frac{1}{k}$$

Spiel D: Kandidaten $\{x_k, \dots, x_n\}$, zähle wie oft ein Element nach x_k gezogen wird, das minimal ist.

$$D^k = \frac{1}{n-k+1} B^{k-1} + \sum_{i=k+1}^n (\frac{1}{n-k+1} D^{i-k})$$

Satz: Sei T ein RST für eine Menge von n Schlüsseln.

1. Die erwartete Laufzeit für Insert, Delete und Lookup ist jeweils $\mathcal{O}(\log(n))$

2. Die erwartete Zahl der Rotationen bei Insert oder Delete ist < 2

Beweis:

1. Kosten für Lookup: $E(|P_k|)$ nach einem x_k

$$\text{Lemma Teil a)} E(|P_k|) = H_k + H_{n-k+1} - 1$$

$$\Rightarrow \leq 2H_n = \mathcal{O}(\ln(n)) = \mathcal{O}(\log(n))$$

Insert, Delete von x_k

$$\text{Kosten } \mathcal{O}(|P_k| + |R_k|) = \mathcal{O}(H_k + H_{n-k+1} - 1 + 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1}) = \mathcal{O}(H_n) = \mathcal{O}(\log(n))$$

2. Erwartete Anzahl an Rotationen:

$$E(|L_k|) + E(|R_k|) = 1 - \frac{1}{k} + 1 - \frac{1}{n-k+1} < 2$$

1.3 Hashing

Speicherung dünn besetzter Tabellen / sparse tables

Spezielle Wörterbücher, da Schlüssel ganze Zahlen. Es werden keine Vergleiche (\leq, \geq etc.) auf der Schlüsselmenge durchgeführt. Es gibt keine lineare Ordnung.

Genauer: Verwaltet Schlüsselmenge $S \subseteq \{0, \dots, N-1\}$ mit eventuell dazugehörigen Daten (Satellitendaten).

Operationen: Lookup(x), Insert(x), Delete(x)

Wie immer gilt $n = |S|$, $N = \text{Anzahl aller Schlüssel}$.

Triviale Lösung:

Verwende ein Feld $T[0 \dots N-1]$ (Tafel). Speichere $x \in S$ (mit seinen Daten) an Tafelposition x, d.h. $T[x] \leftarrow x$

Für alle $y \notin S : T[y] \leftarrow \text{null}(-1)$

Lookup(x): return $T[x]$

Problem: Speicherplatz $\mathcal{O}(N)$, Ziel $\mathcal{O}(n)$

Ziel Hashing: Laufzeit $\mathcal{O}(1)$, Speicher $\mathcal{O}(n)$

Hashtable: $T[0 \dots m-1]$ mit m Größe der Tafel $m \ll N$

Hashfunktion: $h : U \rightarrow \{0, \dots, m-1\}$ mit Universum $U = \{0, \dots, N-1\}$. Speichert $x \in S$ an Position $h(x)$

Insert: $T[h(x)] \leftarrow x$ (Daten)

Lookup(x): Teste ob $T[h(x)] == x$

Die Hashfunktion entscheidet, wie die Tabelle kleiner als die Schlüsselmenge werden kann.

Häufig verwendete naive Hashfunktion: $h : x \rightarrow x \bmod m$

Es treten Kollisionen auf, wenn $h(x) = h(y)$ mit $x \neq y$

Kollisionsbehandlung:

1. Hashing mit Verkettung (Kollisionsmengen werden in Liste gespeichert und mit abgefragtem Wert abgeglichen)
Oft $m < n$. Speichere alle Schlüssel $x \in S$ mit $h(x) = i$ in einer Liste $T[i]$. Meist wird als Funktion der einfache Modulo verwendet.

Lookup(x): Durchsuche Liste von $T[h(x)]$ linear. $\mathcal{O}(1 + |T[h(x)]|)$, worstcase $\mathcal{O}(n)$

Erwartete Kosten: $\mathcal{O}(1 + \frac{n}{m})$ (Übung), Belegungsfaktor $\beta = \frac{n}{m}$ (Erwartete Länge einer Liste $T[x]$)

Insert(x): Falls Lookup(x)=null füge x an erste Stelle von $T[h(x)]$ ein.

Delete(x): Entfernt x aus der Liste $T[h(x)]$

Verbesserung: Immer wenn $\beta > 4$ wird, verdopple die Tafelgröße. 1 sehr teures Insert →, im Schnitt weiter $\mathcal{O}(1)$

Bei Delete und kleinem β kann Tabelle halbiert werden. 1 sehr teures Delete.

2. Hashing mit offener Adressierung (Ausprobieren einer Folge von Positionen)

Voraussetzung: $n \leq m$ und damit $\beta \leq 1$

Idee: Folge von Hashfunktionen $h_0, h_1, \dots: h_i(x) = (f(x) + i * g(x)) \bmod m$

Mit f,g Hashfunktionen $U \rightarrow \{0, \dots, m-1\}$

f(x) gibt die Startposition an, g(x) verschiebt diese.

$h(x)=1$ heißt Linear Probing.

Falls belegt probiere $h_1(x), h_2(x), \dots$, bis freie Stelle gefunden.

Der Status der Positionen kann in einem zweiten Feld $\text{status}[0..m-1]$ gespeichert werden (frei, besetzt, gelöscht).

Lookup(x): Durchsuche Folge $T[h_0(x)], T[h_1(x)] \dots$ bis x gefunden, oder freie Position (dann ist x nicht enthalten)

Delete(x): Lookup(x) , markiere Position als frei. Problem: Elemente dahinter unerreichbar.

Lösung: gelöscht flag im Statusarray. Lookup übergeht diesen und hält nicht, Insert erkennt es als freies Feld.

3. Perfektes Hashing (keine Kollision durch injektive Funktion) Voraussetzung $n \leq m$

$S \subseteq \{0, \dots, N-1\}$ $n = |S|$ verwende Tafel der Größe $m \geq n$ und $m = \mathcal{O}(n)$

Statisch: S ist fest, nur 2 Operationen. Init(S) Konstruktor, Lookup(x)

Ziel: Tafelgröße $s = \mathcal{O}(n)$, Hashfunktion injektiv auf S

Gegeben $S \subseteq \{0, \dots, N-1\}$ und Tafel $T[-, \dots, s-1]$

Injektive Funktion $h: \{0, \dots, N-1\} \rightarrow \{0, \dots, s-1\}$ injektiv auf S

Idee: Verwende ein randomisiertes Verfahren, d.h. wähle eine zufällige Funktion aus den Kandidaten.

Originalarbeit: Storing a Sparse Table with $\mathcal{O}(1)$ worst-case Access Time

Verwende zweistufiges Hashing Schema (injektiv), Auswahl der Funktion durch Randomisierung

1. Schritt: Funktion (muss noch nicht injektiv sein) bildet auf eine Liste von Buckets (W_0, \dots, W_{s-1}) $W_i = \{x \in S | h(x) = i\}$ ab, mit s der Größe der ersten Stufe.

2. Schritt: Für jedes W_i gibt es eine eigene Hashfunktion h_i , die jeweils auf eine weitere Tafel der Größe s abbilden. h_i ist injektiv auf W_i . Für jedes W_i gibt es eine eigene 2. Tafel $\{m_0, \dots, m_{s-1}\}$. Dadurch gibt es auf der zweiten Stufe quadratische Tafelgröße.

Man findet "leicht" eine Funktion h der ersten Stufe mit $\sum_{i=0}^{s-1} |W_i|^2 = \mathcal{O}(n)$

Für Tafeln quadratischer Größe findet man relativ leicht injektive Hashfunktion. Sei p eine Primzahl mit $p > N$.

$U = \{0, \dots, N-1\}, S \subseteq U, n = |S|, s \in \mathbb{N}$ Tafelgröße

Betrachte folgende Hashfunktionen h_1, \dots, h_{p-1}

$h_k: \{0, \dots, N-1\} \rightarrow \{0, \dots, s-1\}$

$h_k(x) = (k * x \bmod p) \bmod s$

Jede Funktion $h_k, 1 \leq k \leq p-1$ verteilt die Menge S auf s Buckets:

W_0^k, \dots, W_{s-1}^k , d.h. genauer: $W_i^k = \{x \in S | h_k(x) = i\}$

Lemma1: Für jede Menge $S \subseteq \{0, \dots, N-1\}$ mit $|S| = n$ gilt:

$\exists k, 1 \leq k \leq p-1$ mit $\sum_{i=0}^{s-1} (\binom{|W_i^k|}{2}) < \frac{n^2}{s}$

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$: Anzahl der k-elementigen Teilmengen einer Menge der Größe n

$\binom{n}{2} = \frac{n \cdot n - 1}{2}$

hier: $\binom{|W_i^k|}{2}$: Anzahl aller $\{x, y\} x \neq y, x, y \in S$, die im selben Bucket W_i^k landen. (#Kollisionen) Beweis:

Behauptung: $\sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < (p-1) \frac{n^2}{s}$ (Beweis später)

Daraus folgt Lemma1. Indirekt gilt Lemma 1 nicht $\Rightarrow \forall 1 \leq k \leq p-1 : \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} \geq \frac{n^2}{s} \Rightarrow \sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} \geq (p-1) \frac{n^2}{s}$ Widerspruch zu Behauptung

Folgerung 1: Für $s = n$ (d.h. Tafelgröße n) folgt aus Lemma1: $\exists k, 1 \leq k \leq p-1 : \sum_{i=0}^{s-1} (|W_i^k|^2) < 3n$

Beweis: Betrachte Lemma1 für $s=n$

$\exists 1 \leq k \leq p-1 : \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < n$

$\sum_{i=0}^{s-1} \frac{|W_i^k| * (|W_i^k| - 1)}{2} < n$

$\sum_{i=0}^{s-1} |W_i^k|(|W_i^k| - 1) < 2n$

$$\sum_{i=0}^{n-1} (|W_i^k|^2 - |W_i^k|) < 2n$$

$$\sum_{i=0}^{n-1} |W_i^k|^2 < 2n + \sum_{i=0}^{n-1} |W_i^k| \text{ wobei } \sum_{i=0}^{n-1} |W_i^k| = n \text{ also } |S|$$

Folgerung 2: Für $s = n^2$ folgt aus Lemma 1: $\exists 1 \leq k' \leq p-1$ sodass die Hashfunktion $h_{k'} : x \rightarrow (k' * x \bmod p) \bmod n^2$ die injektiv auf S ist, d.h. $|W_i^{k'}| \leq 1$ für $i = 0, \dots, n^2-1$

Für quadratische Tafelgrößen existiert eine perfekte Hashfunktion $h_{k'}$

Beweis: Betrachte Lemma 1 für $s = n^2$

$$\exists 1 \leq k' \leq p-1 \text{ mit } \sum_{i=0}^{n-1} \binom{|W_i^{k'}|}{2} < \frac{n^2}{2} = 1 \Rightarrow \sum_{i=0}^{n^2-1} \binom{|W_i^{k'}|}{2} = 0 \Rightarrow \forall 0 \leq i \leq n^2-1 : |W_i^{k'}| \leq 1 \Rightarrow \#Kollisionen = 0$$

Vermeidung des quadratischen Speicherplatzes durch ein 2-Stufiges hashing-Schema.

1. Stufe: Verwende Tafelgröße $s=n$ und wähle ein k gemäß der Folgerung d.h. $\sum_{i=0}^{n-1} |W_i^k|^2 < 3n$

Die Hashfunktion $h_k(x) : x \rightarrow (kx \bmod p) \bmod n$ verteilt s auf eine Tafel der Größe n, sodass die Summe der Quadrate der Bucketgrößen kleiner ist als $3n$.

2. Stufe: Für jedes nicht-leere Bucket W_i^k der 1. Stufe verwende eine Tafel der Größe $s_i = |W_i^k|^2$ und wähle k_i gemäß Folgerung 2. Genauer: Für $i=0, \dots, n-1$ wähle ein k_i sodass $h_{k_i}(x) : x \rightarrow (k_i x \bmod p) \bmod s_i$ injektiv auf W_i^k ist.

Gesamtbedarf: 1. Stufe: Platz n, 2. Stufe: $\sum_{i=0}^{n-1} |W_i^k|^2 < 3n$ ungefähr $4n$, genauer später.

Problem: Wie findet man diese injektiven Funktionen, d.h. wie wählt man k und k'

Idee: Erhöhe die Tafelgrößen auf Stufe 1 und Stufe 2 um einen konstanten Faktor. Dann erfüllen mindestens 50% aller k die Bedingungen von Folge 1&2

Beweis der Behauptung:

$$(*) \sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < (p-1) \frac{n^2}{s} \Rightarrow \exists k : \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < \frac{n^2}{s}$$

Mit p Primzahl $p \geq N$ Die Summe ist: Anzahl aller Paare $(l, \{x, y\})$ mit $1 \leq l \leq p-1$, $x, y \in S$, $x \neq y$ und $h_k(x) = h_k(y)$ (Kollision).

Beitrag eines festen Paares $x \neq y$ zu der Summe = Anzahl aller k mit $h_k(x) = h_k(y)$

$$(kx \bmod p - ky \bmod p) \bmod s = 0$$

$$kx \bmod p - ky \bmod p = i * s \quad i \in \mathbb{Z}$$

$k * (x-y) \bmod p \in \{-(p-1), \dots, p-1\}$ Vielfaches von s aus der Menge $\Rightarrow \frac{2*(p-1)}{s}$ verschiedene Gleichungen
Lösen der Gleichung nach k $k = \frac{i*s}{x-y} \bmod p$ Da p eine Primzahl (\mathbb{Z}_p ein Körper) existiert eine wohl-definierte Division (inverse zur Multiplikation) und daher besitzt jede Gleichung höchstens eine Lösung für k .

$$\Rightarrow \text{Der Beitrag von } \{x, y\} x \neq y \text{ zur Summe } \leq \frac{2(p-1)}{s}$$

Wir summieren über alle 2-elementigen Teilmengen $\{x, y\}, x \neq y, x, y \in S$:

$$\text{Anzahl der Teilmengen } (*) \leq \binom{n}{2} * \frac{2(p-1)}{s} = \frac{n(n-1)}{2} * \frac{2(p-1)}{s} \leq n^2 \frac{p-1}{s}$$

Implementierung Beispiel:

3 Felder (1. Stufe) $W[0 \dots n-1]$ mit $W[i]$ Pointer auf Bucket 2. Stufe.

$\text{size}[0 \dots n-1]$ Anzahl aller Elemente in W_i^k

$K[0 \dots n-1]$ mit $K[i] = k_i$ k-Werte der 2. Stufe.

Variable $k_0 = k$ der ersten Stufe

Für 2. Stufe: n Tafeln B_i mit $i = 0 \dots n-1$ und $B_i[0 \dots \text{size}[i]^2 - 1]$

Platzbedarf: $\text{size} + K + W + k_0 + B_i = 3n + 1 + \sum_{i=0}^{n-1} \text{size}[i]^2 \leq 6n + 1 = \mathcal{O}(n)$

Speichere $x \in S$ wie folgt: $i \leftarrow h_{k_0}(x)$, $j \leftarrow h_{K[i]}(x)$, $W[i][j] \leftarrow x$

Mit $h_{k_0} = ((k_0 * x) \bmod p) \bmod n$ und $h_{K[i]}(x) = ((K[i] * x) \bmod p) \bmod \text{size}[i]^2$

Lookup: Teste ob $W[i][j] == x$

Bemerkung: Für $\text{size}[i]=0$ kein B_i auf 2. Stufe, $W[i]=\text{null}$; Für $\text{size}[i]=1$ keine neue Hashfunktion $K[i]$, speichere direkt.

Für kleine $\text{size}[i]$ konstant kann man Verkettung anstelle von Hashfunktion+Tabelle nehmen. Aufbauzeit: $\mathcal{O}(n * N)$ für Garantie eines k (alle k ausprobieren)

Randomisierung: Wir brauchen viele geeignete k 's, dann ist mit hoher Wahrscheinlichkeit ein zufällig gewähltes gut.

Folgerung 3: Für $s = n$ in Lemma 1 gilt für mindestens die Hälfte aller k $1 \leq k \leq p-1$ $\sum_{i=0}^n |W_i^k|^2 < 5n$

Beweis: $\sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < (p-1) \frac{n^2}{s}$

$$\sum_{k=1}^{p-1} \sum_{i=0}^{s-1} \binom{|W_i^k|}{2} < (p-1)n$$

Anhand der Behauptung halbte aller ks: $\sum_{i=0}^{n-1} \binom{|W_i^k|}{2} < 2n$ denn sonst ist für mehr als die Hälfte der Wert ≥ 2 (2n ist ungefähr der Mittelwert der W_i^k 's.

$$\Rightarrow \sum_{i=0}^{n-1} \frac{|W_i^k|(|W_i^k|-1)}{2} < 2n$$

$$\sum_{i=0}^{n-1} |W_i^k|^2 - \sum_{i=0}^{n-1} |W_i^k| < 4n \text{ Da } \sum_{i=0}^{n-1} |W_i^k| = n: \sum_{i=0}^{n-1} |W_i^k|^2 < 5n$$

Folgerung 4: Für $s = 2n^2$ (Lemma 1) gilt für mindestens die Hälfte aller $k' 1 \leq k' \leq p-1$: $h_{k'}(x) \rightarrow (k'x \bmod p) \bmod 2n^2$ ist injektiv auf S.

$$\text{Beweis: } s = 2n^2: \sum_{k=0}^{p-1} \sum_{i=0}^{2n^2-1} \binom{|W_i^k|}{2} < (p-1) \frac{n^2}{2n^2} \Rightarrow < \frac{1}{2}(p-1)$$

Daraus folgt, dass für mindestens die Hälfte aller $k 1 \leq k \leq p-1$ gilt: $\sum_{i=0}^{2n^2-1} \binom{|W_i^k|}{2} = 0$ (das entsprechende h_k ist injektiv).

Platzbedarf: 1. Stufe: $3n + 1$, 2. Stufe: $\sum_{i=0}^{n-1} 2|W_i^k|^2 = 2 \sum_{i=0}^{n-1} |W_i^k|^2 < 10n$. Insgesamt $13n+1 = \mathcal{O}(n)$

Randomisierte Berechnung der k-Werte. Mindestens die Hälfte sind geeignet. Dann ist zufälliges k mit Wahrscheinlichkeit $\frac{1}{2}$ ok. \Rightarrow Erwartete Anzahl von zufälligen Ziehungen bis ein k gefunden ist ist 2. Erwartete Aufbauzeit $\mathcal{O}(n)$

Zusammenfassung (Perfect Hashing): Jede Menge $S \subseteq \{0, \dots, N-1\}$ mit $|S| = n$ kann so abgespeichert werden, dass gilt:

- 1: Platzbedarf ist $\mathcal{O}(n)$ ($13n$)
- 2: Erwartete Aufbauzeit ist $\mathcal{O}(n)$
- 3: Zugriff (Lookup) in Zeit $\mathcal{O}(1)$ im worst-case

Dynamisierung (Insert/Delete) ist möglich (Dynamic perfect hashing)

Bei Kollisionen (unwahrscheinlich) muss Tafels 2. Stufe neu gebaut werden (Rehashing)

1.4 Priority Queues

Definition des Datentyps: Priority Queue PQ speichert Paare (p, i) von Prioritäten und Informationen $p \in P, i \in I$

z.B. P ganze Zahlen, I Knoten eines Graphen

Operationen: PQ.insert(p, i) fügt das Paar (p, i) ein. PQ.findmin() liefert ein Paar mit minimaler Priorität. PQ.delmin() entfernt das Paar PQ.findmin(). PQ.decrease_p(x, q) vermindert die Priorität des Paars $x=(p, i)$ auf q d.h. (q, i) mit $q < p$

1.4.1 Mögliche Implementierung

1. Unsortierte Liste: Liste speichert Wert und Priorität, Insert/Decrease= $\mathcal{O}(1)$, Find= $\mathcal{O}(n)$
2. Sortierte Liste (Suchbaum)
3. Fibonacci Heap

1.4.2 Amortisierte Analyse Fibonacci Heap

Abschätzung einer beliebigen Folge von Operationen auf einer Datenstruktur D: $D_0 \rightarrow^{op_0} D_1 \rightarrow^{op_1} D_2 \dots$

Die amortisierten Kosten sind die durchschnittlichen Kosten einer beliebigen Operation. (Hier Gesamtkosten / n)

Beispiel: Binärzähler mit den Operationen Init (Setze auf 0) und Increase (erhöhe um 1)

Datenstruktur: Bitstring ... $a_3a_2a_1a_0$ mit $a_i \in \{0, 1\}$

$$\text{Zählerwert} = \sum_{i \geq 0} a_i 2^i$$

Operationen: Init: Erzeuge Liste der Länge 1 mit $a_0 = 0$. Kosten $\mathcal{O}(1)$

Increment:

$$a_0 \leftarrow a_0 + 1$$

$$i = 0$$

while $a_i = 2$ do

$$a_i = 0; a_{i+1} \leftarrow a_{i+1} + 1; i \leftarrow i + 1;$$

done

Increment Laufzeit: $\mathcal{O}(1 + \text{Anzahl Überträge})$ Laufzeit einer Folge Init und n Increment: $\mathcal{O}(n * \log(n))$ (worst case), da die Zahl $\log(n)$ Bits hat.

Da so hohe Überträge sehr selten sind eher $\mathcal{O}(n)$

1.4.3 Amortisierte Analyse: Potentialmethode

Allgemein: Datenstruktur D, Potential (Funktion $pot : D \rightarrow R_0^+$)

Operation: $op : D' \rightarrow^{op} D''$

Definition: $T_{\text{Tats}}(op) :$ Ausführungszeit einer Operation (Tatsächliche Kosten)

$$T_{\text{Amort}}(op) = T_{\text{Tats}}(op) + pot(D'') - pot(D')$$

$$\text{D.h.: } T_{\text{Amort}}(op) = T_{\text{Tats}}(op) + \Delta pot$$

Betrachte nun eine Folge von n Operationen: $D_0 \rightarrow^{op_0} D_1 \rightarrow^{op_1} D_2 \rightarrow^{op_n} D_n$

Dann gilt: $\sum_{i=1}^n T_{\text{Tats}}(op_i) = \sum_{i=1}^n T_{\text{Amort}}(op_i) + pot(D_0) - pot(D_n)$

Spezialfall: $pot(D_0) = 0; pot(D_i) \geq 0 \Rightarrow \sum_{i=1}^n T_{\text{Tats}}(op_i) \leq \sum_{i=1}^n T_{\text{Amort}}(op_i)$ Anwendung auf Zähler ($\text{INIT} \rightarrow D_0 \rightarrow^{increment} D_1 \dots$):

Problem: Wähle gut Potentialfunktion, die eine gute Schranke liefert.

Hier: $pot =$ Anzahl aller 1 im Bitstring. Bedingung des Spezialfalls erfüllt.

Betrachte nun T_{Tats} und T_{Amort}

Wenn der Zähler nun folgende Form hat: $xxx0111$ mit der ersten Null von rechts aus gesehen und k Einsen davor, dann ist:

$$1. T_{\text{Tats}}(incr) = \mathcal{O}(1+k)$$

$$2. \Delta pot = 1 - k$$

$$\text{Also: } T_{\text{Tats}}(incr) = 1 + k + (1 - k) = 2 = \mathcal{O}(1)$$

\Rightarrow Potentialmethode Gesamtkosten $2n = \mathcal{O}(n)$

1.4.4 Fibonacci Heaps

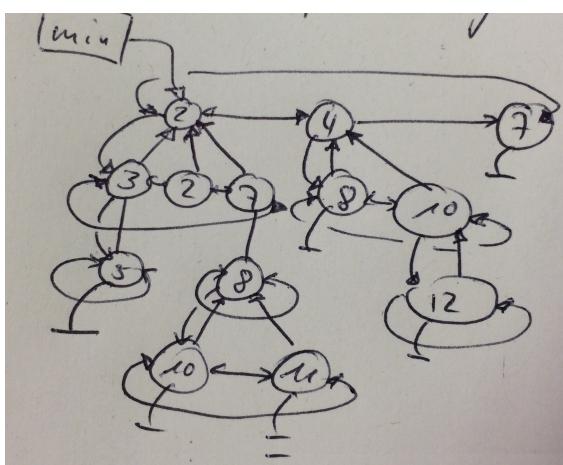
Warteschlangen werden realisiert durch eine Menge sogenannter "Heap ordered Trees".

Ein Baum T heißt "Heap ordered", wenn seine Knoten mit Prioritäten beschriftet sind, sodass für alle Knoten $v \neq \text{Wurzel}$ gilt: $\text{prio}(v) \geq \text{prio}(\text{vater}(v))$.

Wir benötigen einige Pointer in den Knoten: Vater, Verweise auf 1. Kind, alle Kinder eines Knoten in zyklischer doppeltverketteten Liste (2 Verweise im Knoten), und Daten

Mögliche Klasse:

```
class node{
    P prio;
    I indo;
    node parent, left_sib, right_sib;
    bool mark;
    int rank
}
```



Auch Wurzel ist in einer zyklischen Liste (Wurzelliste).

Realisierung der Operationen:

Konstruktor(Init) $\mathcal{O}(1)$: min=0

PQ.findmin() $\mathcal{O}(1)$: Zugriff über min_pointer

PQ.insert(p,i) $\mathcal{O}(1)$: Erzeuge neuen Baum, der nur aus 1 Knoten besteht. Füge v in die Wurzelliste dieses Baums ein, v.prio = p, v.info = i. Falls $p < \min_{v \in \text{Wurzeln}} \text{prio}$ dann $\min = v$

PQ.delmin(): Bringt Fib.-Heap in eine besondere Form: Alle Wurzeln sollen paarweise unterschiedliche Ränge haben (# Kinder).

delmin im Detail:

1. $v \leftarrow \min$
2. Lösche v aus der Wurzelliste (doppelt verkettet) und füge alle Kinder v in die Wurzelliste ein. (Hier entstehen gleiche Ränge. $\mathcal{O}(\text{rang}(v))$)
3. Eliminiere Mehrfach-Vorkommen des gleichen Rangs:
while $\exists 2$ Wurzeln v_1, v_2 gibt, sodass $v_1 \neq v_2$ und $\text{rang}(v_1) = \text{rang}(v_2)$
do: If $v_1.\text{prio} \leq v_2.\text{prio}$, sonst vertausche ; Verschmelze(v_1, v_2) (Mache v_2 zum Kind von v_1)
(Hänge Baum größerer Priorität an den der kleineren Priorität)
Daraus folgt: Alle Wurzeln haben verschiedenen Rang $\Rightarrow \# \text{Wurzeln} \leq \max \text{Rang}$.
Kosten: $\mathcal{O}(\max \text{Rang})$
4. Lineare Suche nach Minimum

Lemma: Schritt 3 hat Laufzeit $T_{\text{Tats}} \mathcal{O}(\max \text{Rang} + \# \text{Verschmelzungen})$

Beweis: durch Algorithmus

Idee: Verwende Feld (Tafel) Rank[0,...,maxRang]. Speichere in Rank[i] die Wurzel mit Rang=i (unter allen bereits bearbeiteten Wurzeln)

```
for i=0 to maxRang do Rank[i] = null od
forall wurzel v in Wurzelliste do
    while Rank[rang(v)] != null do
        vtmp = Rank[rang(v)]
        Rank[rang(v)] = null
        v = Verschmelze(v, vtmp)
    done
    Rank[rang(v)] = v
done
```

Laufzeit: Sei $\# W_0 = \#$ Wurzeln vor z.2; $\# W_1 = \#$ Wurzeln nach z.2; $\# V = \#$ Verschmelzungen

Dann gilt:

1. Laufzeit von z.2 $\mathcal{O}(\max \text{Rang} + \# W_0 + \# V)$
 2. $\# W_0 = \# W_1 + \# V$ Weil jede Verschmelzung eine Wurzel eliminiert
 3. Am Ende ist $\# W_1 \leq \max \text{Rang}$ da alle Ränge verschieden
- $$\Rightarrow \mathcal{O}(\max \text{Rang} + \# W_0 + \# W_1) = \mathcal{O}(\max \text{Rang} + \# W_1 + \# V + \# W_1) = \mathcal{O}(\max \text{Rang} + \# W_1 + \# V) = \mathcal{O}(\max \text{Rang} + \# V)$$

Da $\# W_1 \leq \max \text{Rang}$. Es bleibt abzuschätzen was maxRang und $\# V$ ist.

Wir schätzen den maximalen Rang ab. Beobachtung Insert,Delmin erzeugen spezielle Bäume (binomische Bäume).

Folge von Bäumen: T_0, \dots, T_i mit T_0 ein Knoten, T_{i+1} ist T_i mit einem weiteren T_i als zusätzliches Kind.

Der Maximale Rang ist an der Wurzel. Anzahl der Knoten verdoppelt sich mit steigendem i, Rang erhöht sich um 1.

Lemma: $\max \text{Rang} = \log(n)$ in binomischen Bäumen, wobei n die Anzahl der Knoten ist.

Beweis: 1. Die Wurzel von T_i hat Rang i, T_i hat 2^i Knoten (verdoppelt sich in jedem Schritt $i \rightarrow i+1$

\Rightarrow Wurzel hat Rang $\log(n)$.

2. Wurzel hat maximalen Rang in T_i , da alle Nachkommen Wurzen von T_j sind, mit $j < i$. 2. Folgt aus 1. Fibonacci-Heap ist eine Menge von binomischen Bäumen. Nach Lemma gilt $\max \text{Rang} \leq \log(n)$ wobei n Anzahl der Knoten ist.

Amortisierte Analyse:

Potential = Anzahl der Wurzeln.

$T_{\text{amort}}(op) = T_{\text{Tats}}(op) + \Delta \text{pot}$

Betrachte beliebige Folge von Operationen. Dann sind:

create = $\mathcal{O}(1)$, finMin = $\mathcal{O}(1)$, da $T_{\text{tats}} = \mathcal{O}(1)$ und $\Delta pot = 0$.

insert: $T_{\text{tats}} = \mathcal{O}(1)$ und $\Delta pot = 1 \Rightarrow T_{\text{amort}} = \mathcal{O}(1)$

delmin: $T_{\text{tats}} = \mathcal{O}(\max Rang + \#V)$. $\Delta pot \leq \max Rang - \#V$, wobei die Kinder von v zu Wurzeln werden und jede Verschmelzung eine Wurzel entfernt

$$T_{\text{amort}} = T_{\text{tats}} + \Delta pot \leq \max Rang + \#V + \max Rang - \#V = 2\max Rang = \mathcal{O}(\log(n))$$

Bemerkung: Amortisierte Analyse kann auch modelliert werden durch sogenannte "coin operated machines".

Das Potential wird dabei als eine Art Konto angesehen.

Eine beliebige Folge von n_1 Inserts, n_2 FindMins, n_3 DelMins und 1 Create hat Gesamtkosten $\mathcal{O}(n_1 + n_2 + n_3 * \log(n))$

Worstcase Kosten von delMin sind $\mathcal{O}(n)$

Wir behandeln nun PQ.decrease(v,q): v Knoten, q neue kleinere Priorität.

Schritte:

1 v.prio = q (verminderung), verletzt u.U. die Heap-Eigenschaft $\mathcal{O}(1)$

cut Lösche den Pointer zum Vaterknoten und füge v in die Wurzelliste ein (und aus Geschwisterliste löschen), Kinderknoten hängen weiterhin an v, setze v.mark=false $\mathcal{O}(1)$

3 Update des min-Pointers für minimale Wurzel $\mathcal{O}(1)$

Achtung: Baum ist nicht mehr binomisch $\rightarrow \log(n)$ Schranke für maxRang ist nicht mehr gültig

Wie kann man den Rang dennoch auf $\log(n)$ beschränken:

Idee: Setze das Löschen von Vaterverweisen in Richtung Wurzel fort, sobald ein zweites Kind entfernt wird:

4 Falls v keine Wurzel, Markiere(Vater(v))

Markiere(v) :

```

while w.parent!=null UND w.mark=true do
    u=w.parent
    Fuege w in Wurzelliste ein
    w.mark = false
    w=u
done
if w.parent!=null then
    w.mark=true
fi
```

Wurzeln sind stets unmarkiert.

Beobachtung: Eine decrease-Operation kann sehr viele cut-Operationen zur Folge haben.

Wie zeigen, dass durch diese Markierungsstrategie der maximale Rang durch $\mathcal{O}(\log(n))$ beschränkt.

Lemma: $\max Rang \leq 1,4404 * \log(n)$

Beweis: Idee: Zeige, dass # Knoten exponentiell mit maxRang wächst (untere Schranke) $\Rightarrow \max Rang = \mathcal{O}(\log(n))$

Sei v beliebiger Knoten mit Rang i. Ordne die Kinder von v nach der Zeit, zu der sie am Knoten v angehängt wurden (Verschmelzung)

Sei w_j das j. Kind.

Behauptung: $Rang(w_j) \geq j - 2$

Beweis: Als w_j Kind von v wurde galt: $Rang(w_j) = j - 1$. w_j hat seitdem höchstens 1 Kind verloren (da er nicht Wurzel ist, Markierungsstrategie) $\Rightarrow Rang(w_j) \leq j - 2$

Sein nun v ein Knoten mit Rang i und $s_i :=$ minimale Zahl von Knoten um Unterbaum mit Wurzel vom Rang i.

$s_0 = 1; s_1 = 2$ trivial. Für $i > 2 : s_i = 2 + s_0 + \dots + s_{i-1} = 2 + \sum_{j=0}^{i-1} s_j$

Betrachte die Folge der Fibonacci-Zahlen $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ ($i \geq 2$)

Es gilt 1: $F_{i+2} = 2 + \sum_{j=2}^i F_j$ für $i \geq 1$

2: $s_i \geq F_{i+2}$ für $i \geq -$

Beweis (Induktion über i): $i = 0 : s_0 = 1 = F_2$

$$i > 0 s_i = 2 + \sum_{j=0}^{i-2} s_j \geq 2 + \sum_{j=0}^{i-2} F_{j+2} = 2 + \sum_{j=1}^i F_j = F_{i+2}$$

Außerdem gilt: $F_{i+1} \geq (\frac{1+\sqrt{5}}{2})^i \geq (1,618)^i$ für $i \geq 0$

$$s_i \geq 1,618^i$$

Dann gilt: $s_r \leq n$ mit r maximaler Rang und n # Knoten, d.h. $1,618^r \leq n \Rightarrow r \leq \log_{1,618}(n) = \frac{\log(n)}{\log(1,618)} =$

$1,4404.. * \log(n)$

Das heißt, dass $\max\text{Rang} \leq 1,4404 * \log(n) = \mathcal{O}(\log(n))$

Amortisierte Analyse der Gesamtkosten einer beliebigen Folge von Insert, Findmin, Delmin, Decrease:

Wir modifizieren das Potential $pot = \#Wurzeln + 2 * \#\text{markierteKnoten}$. Analyse von Insert, Findmin und Delmin bleibt gleich, da sich nichts an den Markierungen ändert.

Decrease-p: $T_{\text{tats}} = \mathcal{O}(k+1)$. Δpot : # Wurzeln erhöht sich um $k+1$, # markierte Knoten nimmt ab um $k-1$

$$\Delta pot = (k+1) + 2 * (1-k) = 3 - k \Rightarrow T_{\text{armort}} = T_{\text{tats}} + \Delta pot = k+1 + (3-k) = 4 = \mathcal{O}(1)$$

(Anmerkung: Das k ist die Anzahl der Knoten, die wegen der vorhandenen Markierung um schlimmsten Fall durchlaufen werden müssen)

Satz: Fibonacci-Heaps unterstützen die PQ-Operationen Insert, Findmin, Decrease-p in amortisierter Zeit $\mathcal{O}(1)$ und Delmin in $\mathcal{O}(\log(n))$

Alternativ: Jede Folge von k_1 Inserts, k_2 Findmins, k_3 Delmins und k_4 Decrease-p kostet insgesamt $\mathcal{O}(k_1 + k_2 + k_3 * \log(n) + k_4)$

1.4.5 Übung

Erwartungswert: Summe aller möglichen Werte x multipliziert mit deren Wahrscheinlichkeit.

Hier: $\sum_{k=1}^{\infty} (k * \frac{1}{2}k)$ Integrierende Reihe mit k =Anzahl der Versuche bis zum Erfolg

6.2: Priority Queues:

a) Doppelt verkettete Listen: (Alternativ mit Pointer auf Minimum)

1. Insert(x, p)= $\mathcal{O}(1)$ bzw $\mathcal{O}(1)$

2. findmin= $\mathcal{O}(n)$ bzw $\mathcal{O}(1)$

3. delmin= $\mathcal{O}(n)$ bzw $\mathcal{O}(n)$

b) Suchbaum: Alle Operationen $\mathcal{O}(\log(n))$. Mit Minpointer hat findmin $\mathcal{O}(1)$

c) Binärer Heap (Heapsort):

1. Findmin (Wurzel)

2. Delmin: Entferne Wurzel (Überschreibe den Wert mit unterem Rechten Blatt, lasse Wurzel Sinken, Entferne Blatt) $\mathcal{O}(\log(n))$

3. Insert:Füge als Blatt ein, lasse es an die Richtige Position steigen $\mathcal{O}(\log(n))$

6.3: $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ für $i \geq 2$

a) $F_{i+2} = 2 + \sum_{j=2}^i F_j$ für $i \geq 2$

$$F_{i+2} = F_{i+1} + F_i = 2 + \sum_{j=2}^{i-1} F_j + F_i = 2 \sum_{j=2}^i F_j$$

$$b) F_{i+1} \geq (\frac{1+\sqrt{5}}{2})^i = \Phi^i$$

Beh.: $\Phi^2 = \Phi + 1$

$$F_{i+1} = F_i + F_{i+1} \geq \Phi^{i-1} + \Phi^{i-1} = \Phi^{i-1}(1 + \Phi) = \Phi^{i-1} * \Phi^2 = \Phi^i$$