Randomisierter Algorithmus für das Ski-Problem

Der Algorithmus besteht nur aus einer Verteilung $\pi = \pi_1, \pi_2, \ldots, \pi_t, \ldots$
Gemäß der Verteilung wird der 'Kauftag' t am Anfang ausgelost.

Randomisierter Algorithmus für das Ski-Problem

Der Algorithmus besteht nur aus einer Verteilung $\pi = \pi_1, \pi_2, \ldots, \pi_t, \ldots$
Gemäß der Verteilung wird der 'Kauftag' t am Anfang ausgelost.

Die beste Verteilung π wird in Form einer reellen Funktion $\pi(t)$ ermittelt.
Die beste Funktion erhält man durch Lösen der Integralfunktion

$$\int_0^T (t + K)\pi(t)dt + \int_T^K T\pi(t)dt = \alpha \cdot T$$

Randomisierter Algorithmus für das Ski-Problem

Der Algorithmus besteht nur aus einer Verteilung $\pi = \pi_1, \pi_2, \ldots, \pi_t, \ldots$
Gemäß der Verteilung wird der 'Kauftag' t am Anfang ausgelost.

Die beste Verteilung π wird in Form einer reellen Funktion $\pi(t)$ ermittelt.
Die beste Funktion erhält man durch Lösen der Integralfunktion

$$\int_0^T (t+K)\pi(t)dt + \int_T^K T\pi(t)dt = \alpha \cdot T$$

Theorem:

Mit der Verteilung

$$\pi_t = \frac{e^{t/K}}{K(e-1)}$$

für die randomisierte Strategie wird Wettbewerbsfaktor

$$\alpha = \frac{e}{e-1} \approx 1.58$$

erreicht.

Erinnerung: Metrischer Raum

Erinnerung: Metrischer Raum

Definition:

Ein **metrischer Raum** besteht aus einer Menge X und einer Distanz-Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ so dass

Erinnerung: Metrischer Raum

Definition:

Ein **metrischer Raum** besteht aus einer Menge X und einer Distanz-Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ so dass

a.

$$d(x, y) = 0 \qquad \Longleftrightarrow \qquad x = y$$

Erinnerung: Metrischer Raum

Definition:

Ein **metrischer Raum** besteht aus einer Menge X und einer Distanz-Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ so dass

a.

$$d(x, y) = 0 \qquad \Longleftrightarrow \qquad x = y$$

b.

$$d(x, y) \quad = \quad d(y, x)$$

Erinnerung: Metrischer Raum

Definition:

Ein **metrischer Raum** besteht aus einer Menge X und einer Distanz-Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ so dass

a.

$$d(x, y) = 0 \qquad \Longleftrightarrow \qquad x = y$$

b.

$$d(x, y) \quad = \quad d(y, x)$$

c.

$$d(x, z) \quad \leq \quad d(x, y) + d(y, z)$$

Erinnerung: Metrischer Raum

Definition:

Ein **metrischer Raum** besteht aus einer Menge X und einer Distanz-Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ so dass

a.

$$d(x, y) = 0 \qquad \Longleftrightarrow \qquad x = y$$

b.

$$d(x, y) \quad = \quad d(y, x)$$

c.

$$d(x, z) \quad \leq \quad d(x, y) + d(y, z)$$

d heißt eine **Metrik über** X.

Das k-Server Problem

- (X, d) sei ein endlicher metrischer Raum.
- k Server stehen anfangs auf irgendwelchen Elementen aus X

Das k-Server Problem

- (X, d) sei ein endlicher metrischer Raum.
- k Server stehen anfangs auf irgendwelchen Elementen aus X
- **Eingabe:** $\sigma = r_1, r_2, \ldots, r_t, \ldots$, eine Folge von Anfragen an den Elementen $r_t \in X$

Das k-Server Problem

- (X, d) sei ein endlicher metrischer Raum.
- k Server stehen anfangs auf irgendwelchen Elementen aus X
- **Eingabe:** $\sigma = r_1, r_2, \ldots, r_t, \ldots$, eine Folge von Anfragen an den Elementen $r_t \in X$
- **Aufgabe:** Wenn $r_t \in X$ ankommt, bewege einen der k Server auf Position r_t (und verschiebe evtl. weitere Server beliebig)

Das k-Server Problem

- (X, d) sei ein endlicher metrischer Raum.
- k Server stehen anfangs auf irgendwelchen Elementen aus X
- **Eingabe:** $\sigma = r_1, r_2, \ldots, r_t, \ldots$, eine Folge von Anfragen an den Elementen $r_t \in X$
- **Aufgabe:** Wenn $r_t \in X$ ankommt, bewege einen der k Server auf Position r_t (und verschiebe evtl. weitere Server beliebig)
- **Ziel:** Minimiere die Gesamtlänge aller Serverbewegungen

Das k-Server Problem

- (X, d) sei ein endlicher metrischer Raum.
- k Server stehen anfangs auf irgendwelchen Elementen aus X
- **Eingabe:** $\sigma = r_1, r_2, \ldots, r_t, \ldots$, eine Folge von Anfragen an den Elementen $r_t \in X$
- **Aufgabe:** Wenn $r_t \in X$ ankommt, bewege einen der k Server auf Position r_t (und verschiebe evtl. weitere Server beliebig)
- **Ziel:** Minimiere die Gesamtlänge aller Serverbewegungen

Work-Function Strategie

Work-Function Strategie

- (Multi)menge der Startpositionen: $A_0 = \{a_1, a_2, \ldots, a_k\}$

Work-Function Strategie

- (Multi)menge der Startpositionen: $A_0 = \{a_1, a_2, \ldots, a_k\}$
- Anfragen: $\sigma = (r_1, r_2, \ldots, r_t, \ldots)$

Work-Function Strategie

- (Multi)menge der Startpositionen: $A_0 = \{a_1, a_2, \ldots, a_k\}$
- Anfragen: $\sigma = (r_1, r_2, \ldots, r_t, \ldots)$
- (Multi)menge von k Orten: M

Work-Function Strategie

- (Multi)menge der Startpositionen: $A_0 = \{a_1, a_2, \ldots, a_k\}$
- Anfragen: $\sigma = (r_1, r_2, \ldots, r_t, \ldots)$
- (Multi)menge von k Orten: M

Sei $w_{t-1}(M)$ die **optimale Gesamtlänge** von Serverbewegungen, um
(1) die Anfragen $r_1, \ldots, r_{t-1}$ zu erfüllen und
(2) die Server von A_0 nach M zu bewegen.

Sei A_{t-1} die Positionen der Server im Algorithmus am Ende von r_{t-1}.

Work-Function Strategie

- (Multi)menge der Startpositionen: $A_0 = \{a_1, a_2, \ldots, a_k\}$
- Anfragen: $\sigma = (r_1, r_2, \ldots, r_t, \ldots)$
- (Multi)menge von k Orten: M

Sei $w_{t-1}(M)$ die **optimale Gesamtlänge** von Serverbewegungen, um
(1) die Anfragen $r_1, \ldots, r_{t-1}$ zu erfüllen und
(2) die Server von A_0 nach M zu bewegen.

Sei A_{t-1} die Positionen der Server im Algorithmus am Ende von r_{t-1}.

Auswahl:

Für r_t wähle den Server auf $a \in A_{t-1}$, der

$$w_{t-1}(A_{t-1} \setminus \{a\} \cup \{r_t\}) + d(a, r_t)$$

minimiert.

Work-Function Strategie (umformuliert)

Sei r_t die nächste Anfrage.

Für jeden Server s mit aktueller Position x_s berechne seinen **Bewegungswert**:

Work-Function Strategie (umformuliert)

Sei r_t die nächste Anfrage.

Für jeden Server s mit aktueller Position x_s berechne seinen **Bewegungswert**:

- Berechne die minimale Gesamtlänge der Serverbewegungen, die alle Anfragen erfüllen bis r_{t-1} und die Server von den Startpositionen bewegen zu:

  $\{$ Menge der aktuellen Serverpositionen $\} \setminus \{x_s\} \cup \{r_t\}$

- Addiere dazu die Distanz $d(x_s, r_t)$.
- Dies ergibt den Bewegungswert von Server s.

Bewege den Server s mit minimalen Bewegungswert von x_s zu r_t.

Auswahl von Experten

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose
  3. Die richtige Entscheidung wird mitgeteilt.
     Allgemeiner: Jeder Experte i erhält einen Kostenwert $c_i^t \in [0, 1]$.

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose
  3. Die richtige Entscheidung wird mitgeteilt.
     Allgemeiner: Jeder Experte i erhält einen Kostenwert $c_i^t \in [0, 1]$.
- **Ziel:** Minimiere die Anzahl unserer falschen Entscheidungen, oder allgemeiner, die Gesamtkosten der gewählten Experten.

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose
  3. Die richtige Entscheidung wird mitgeteilt.
     Allgemeiner: Jeder Experte i erhält einen Kostenwert $c_i^t \in [0, 1]$.
- **Ziel:** Minimiere die Anzahl unserer falschen Entscheidungen, oder allgemeiner, die Gesamtkosten der gewählten Experten.
- Der Wettbewerbsfaktor vergleicht den Online-Algorithmus mit dem **bislang besten Experten**

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose
  3. Die richtige Entscheidung wird mitgeteilt.
     Allgemeiner: Jeder Experte i erhält einen Kostenwert $c_i^t \in [0, 1]$.
- **Ziel:** Minimiere die Anzahl unserer falschen Entscheidungen, oder allgemeiner, die Gesamtkosten der gewählten Experten.
- Der Wettbewerbsfaktor vergleicht den Online-Algorithmus mit dem **bislang besten Experten**
- Anwendungen: Lernalgorithmen, kontinuierliche Vermögensanlage

Auswahl von Experten

- Eine **Folge** von Entscheidungen mit n **Experten**
- In jeder Runde $t = 1, \ldots, T$:
  1. Jeder Experte i gibt eine Prognose ab
  2. Wir wählen einen Experten aus und übernehmen seine Prognose
  3. Die richtige Entscheidung wird mitgeteilt.
     Allgemeiner: Jeder Experte i erhält einen Kostenwert $c_i^t \in [0, 1]$.
- **Ziel:** Minimiere die Anzahl unserer falschen Entscheidungen, oder allgemeiner, die Gesamtkosten der gewählten Experten.
- Der Wettbewerbsfaktor vergleicht den Online-Algorithmus mit dem **bislang besten Experten**
- Anwendungen: Lernalgorithmen, kontinuierliche Vermögensanlage

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i
2. Wähle ein $\varepsilon \in (0, \frac{1}{2}]$

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i
2. Wähle ein $\varepsilon \in (0, \frac{1}{2}]$
3. **repeat** in jeder Runde t

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i
2. Wähle ein $\varepsilon \in (0, \frac{1}{2}]$
3. **repeat** in jeder Runde t
4. Wähle einen Experten zufällig. Experte i hat Wahrscheinlichkeit

$$p_i^t = \frac{w_i^{t-1}}{\sum_{k=1}^{n} w_k^{t-1}} \quad .$$

5. Übernimm Entscheidung des gewählten Experten.

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i
2. Wähle ein $\varepsilon \in (0, \frac{1}{2}]$
3. **repeat** in jeder Runde t
4. Wähle einen Experten zufällig. Experte i hat Wahrscheinlichkeit

$$p_i^t = \frac{w_i^{t-1}}{\sum_{k=1}^{n} w_k^{t-1}} \quad .$$

5. Übernimm Entscheidung des gewählten Experten.
6. Nachdem richtiges Ergebnis und Kosten c_i^t bekannt werden,

Randomisierter Weighted Majority Algorithmus

1. Setze $w_i^0 \leftarrow 1$ für jeden Experten i
2. Wähle ein $\varepsilon \in (0, \frac{1}{2}]$
3. **repeat** in jeder Runde t
4. Wähle einen Experten zufällig. Experte i hat Wahrscheinlichkeit

$$p_i^t = \frac{w_i^{t-1}}{\sum_{k=1}^{n} w_k^{t-1}} \ .$$

5. Übernimm Entscheidung des gewählten Experten.
6. Nachdem richtiges Ergebnis und Kosten c_i^t bekannt werden, aktualisiere die Gewichte

$$w_i^t \leftarrow w_i^{t-1}(1 - \varepsilon c_i^t) \ .$$

7. **until** $t = T$ und Zeitspanne vorbei

Amortisierte Kostenanalyse

- Eine Folge von Operationen werden ausgeführt (online oder offline)

Amortisierte Kostenanalyse

- Eine Folge von Operationen werden ausgeführt (online oder offline)
- Operationen mit hohen Kosten kommen (mit Sicherheit) seltener vor

Amortisierte Kostenanalyse

- Eine Folge von Operationen werden ausgeführt (online oder offline)
- Operationen mit hohen Kosten kommen (mit Sicherheit) seltener vor
- Die Gesamtkosten von n aufeinanderfolgenden Operationen sind kleiner als n-mal die Worst-Case Kosten

Amortisierte Kostenanalyse

- Eine Folge von Operationen werden ausgeführt (online oder offline)
- Operationen mit hohen Kosten kommen (mit Sicherheit) seltener vor
- Die Gesamtkosten von n aufeinanderfolgenden Operationen sind kleiner als n-mal die Worst-Case Kosten
- Die **durchschnittlichen Kosten einer Operation** in einer Folge von Operationen sind kleiner als ihre Worst-Case Kosten.

Amortisierte Kostenanalyse

- Eine Folge von Operationen werden ausgeführt (online oder offline)
- Operationen mit hohen Kosten kommen (mit Sicherheit) seltener vor
- Die Gesamtkosten von n aufeinanderfolgenden Operationen sind kleiner als n-mal die Worst-Case Kosten
- Die **durchschnittlichen Kosten einer Operation** in einer Folge von Operationen sind kleiner als ihre Worst-Case Kosten.
- Die durchschnittlichen Kosten einer Operation nennen wir **amortisierte Kosten** der Operation.

Amortisierte Laufzeit

Definition:

- Die **amortisierte Laufzeit von n Operationen** ist (eine obere Schranke für) die Gesamtlaufzeit der n aufeinanderfolgenden Operationen.

Amortisierte Laufzeit

Definition:

- Die **amortisierte Laufzeit von n Operationen** ist (eine obere Schranke für) die Gesamtlaufzeit der n aufeinanderfolgenden Operationen.

- Die **amortisierte Laufzeit einer Operation** ist (eine obere Schranke für) die durchschnittliche Laufzeit pro Operation über n aufeinanderfolgende Operationen.

Potentialfunktion-Argument

Theorem:

Angenommen es gelten für eine Funktion Φ und für die $Kosten_i$ der i-ten Operation

Potentialfunktion-Argument

Theorem:

Angenommen es gelten für eine Funktion Φ und für die $Kosten_i$ der i-ten Operation

- $AmortisierteKosten_i = WirklicheKosten_i + \Phi(i) - \Phi(i-1)$

Potentialfunktion-Argument

Theorem:

Angenommen es gelten für eine Funktion Φ und für die $Kosten_i$ der i-ten Operation

- $AmortisierteKosten_i = WirklicheKosten_i + \Phi(i) - \Phi(i-1)$
- $\Phi(0) = 0$ und

Potentialfunktion-Argument

Theorem:

Angenommen es gelten für eine Funktion Φ und für die $Kosten_i$ der i-ten Operation

- $AmortisierteKosten_i = WirklicheKosten_i + \Phi(i) - \Phi(i-1)$
- $\Phi(0) = 0$ und
- $\Phi(i) \geq 0$,

Potentialfunktion-Argument

Theorem:

Angenommen es gelten für eine Funktion Φ und für die $Kosten_i$ der i-ten Operation

- $AmortisierteKosten_i = WirklicheKosten_i + \Phi(i) - \Phi(i-1)$
- $\Phi(0) = 0$ und
- $\Phi(i) \geq 0$,

dann sind die **amortisierten Gesamtkosten mindestens so groß wie die wirklichen Gesamtkosten**.

Beweis Potentialfunktion

Beweis des Theorems:

$$\sum_{i=1}^{n} AmortisierteKosten_i = \sum_{i=1}^{n} (WirklicheKosten_i + \Phi(i) - \Phi(i-1)) =$$

Beweis Potentialfunktion

Beweis des Theorems:

$$\sum_{i=1}^{n} AmortisierteKosten_i = \sum_{i=1}^{n} (WirklicheKosten_i + \Phi(i) - \Phi(i-1)) =$$

$$\left(\sum_{i=1}^{n} WirklicheKosten_i \right) + \Phi(n) - \Phi(n-1) + \Phi(n-1) - \ldots$$

$$\ldots - \Phi(1) + \Phi(1) - \Phi(0) =$$

Beweis Potentialfunktion

Beweis des Theorems:

$$\sum_{i=1}^{n} AmortisierteKosten_i = \sum_{i=1}^{n} (WirklicheKosten_i + \Phi(i) - \Phi(i-1)) =$$

$$\left(\sum_{i=1}^{n} WirklicheKosten_i \right) + \Phi(n) - \Phi(n-1) + \Phi(n-1) - \ldots$$

$$\ldots - \Phi(1) + \Phi(1) - \Phi(0) =$$

$$\left(\sum_{i=1}^{n} WirklicheKosten_i \right) + \Phi(n) \geq \sum_{i=1}^{n} WirklicheKosten_i$$

Beweis Potentialfunktion

Beweis des Theorems:

$$\sum_{i=1}^{n} AmortisierteKosten_i = \sum_{i=1}^{n} (WirklicheKosten_i + \Phi(i) - \Phi(i-1)) =$$

$$\left(\sum_{i=1}^{n} WirklicheKosten_i \right) + \Phi(n) - \Phi(n-1) + \Phi(n-1) - \ldots$$

$$\ldots - \Phi(1) + \Phi(1) - \Phi(0) =$$

$$\left(\sum_{i=1}^{n} WirklicheKosten_i \right) + \Phi(n) \geq \sum_{i=1}^{n} WirklicheKosten_i$$

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

Operationen: $\mathit{Insert}(x)$, $\mathit{Lookup}(x)$, $\mathit{Delete}(x)$

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

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

Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots$ eine Folge von Operationen

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

Operationen: $\mathit{Insert}(x)$, $\mathit{Lookup}(x)$, $\mathit{Delete}(x)$

Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots$ eine Folge von Operationen

$$\mathit{WirklicheKosten}_i = \begin{cases} 1 & \text{falls keine Reorganisation} \\ m & \text{falls Reorganisation gemacht wird} \end{cases}$$

Reorganisation:

λ : Auslastungsfaktor

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

Operationen: $\mathit{Insert}(x)$, $\mathit{Lookup}(x)$, $\mathit{Delete}(x)$

Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots$ eine Folge von Operationen

$$\mathit{WirklicheKosten}_i = \begin{cases} 1 & \text{falls keine Reorganisation} \\ m & \text{falls Reorganisation gemacht wird} \end{cases}$$

Reorganisation:

λ : Auslastungsfaktor

Wenn $\lambda \geq 1$, füge alle Schlüssel in eine Tabelle doppelter Größe ein

Dynamische Hashtabelle

Szenario:

Sei m die aktuelle Tabellengröße.

Operationen: $\mathit{Insert}(x)$, $\mathit{Lookup}(x)$, $\mathit{Delete}(x)$

Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots$ eine Folge von Operationen

$$\mathit{WirklicheKosten}_i = \begin{cases} 1 & \text{falls keine Reorganisation} \\ m & \text{falls Reorganisation gemacht wird} \end{cases}$$

Reorganisation:

λ : Auslastungsfaktor

Wenn $\lambda \geq 1$, füge alle Schlüssel in eine Tabelle doppelter Größe ein

Wenn $\lambda \leq 1/4$, füge alle Schlüssel in eine Tabelle halber Größe ein

Binomische Heaps

Definition:

Ein **binomischer Heap** ist eine Menge von binomischen Bäumen so dass

Binomische Heaps

Definition:

Ein **binomischer Heap** ist eine Menge von binomischen Bäumen so dass

a. für jedes $i = 0, 1, 2, \ldots$ gibt es höchstens einen Baum B_i,

Binomische Heaps

Definition:

Ein **binomischer Heap** ist eine Menge von binomischen Bäumen so dass

a. für jedes $i = 0, 1, 2, \ldots$ gibt es höchstens einen Baum B_i,

b. jeder Knoten speichert einen Schlüssel und

Binomische Heaps

Definition:

Ein **binomischer Heap** ist eine Menge von binomischen Bäumen so dass

a. für jedes $i = 0, 1, 2, \ldots$ gibt es höchstens einen Baum B_i,

b. jeder Knoten speichert einen Schlüssel und

c. in jedem Baum gilt die Heap-Ordnung: der Schlüssel des Elternknotens ist kleinergleich den Schlüsseln seiner Kinder.

Binomische Heaps

Definition:

Ein **binomischer Heap** ist eine Menge von binomischen Bäumen so dass

a. für jedes $i = 0, 1, 2, \ldots$ gibt es höchstens einen Baum B_i,

b. jeder Knoten speichert einen Schlüssel und

c. in jedem Baum gilt die Heap-Ordnung: der Schlüssel des Elternknotens ist kleinergleich den Schlüsseln seiner Kinder.

Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x

Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x
2. Setze $i \leftarrow 0$

Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x
2. Setze $i \leftarrow 0$
3. **while** es gibt zwei Bäume B_i im Heap **do**

Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x
2. Setze $i \leftarrow 0$
3. **while** es gibt zwei Bäume B_i im Heap **do**
4. Vereinige die beiden B_i so dass die Heap-Ordnung erhalten bleibt

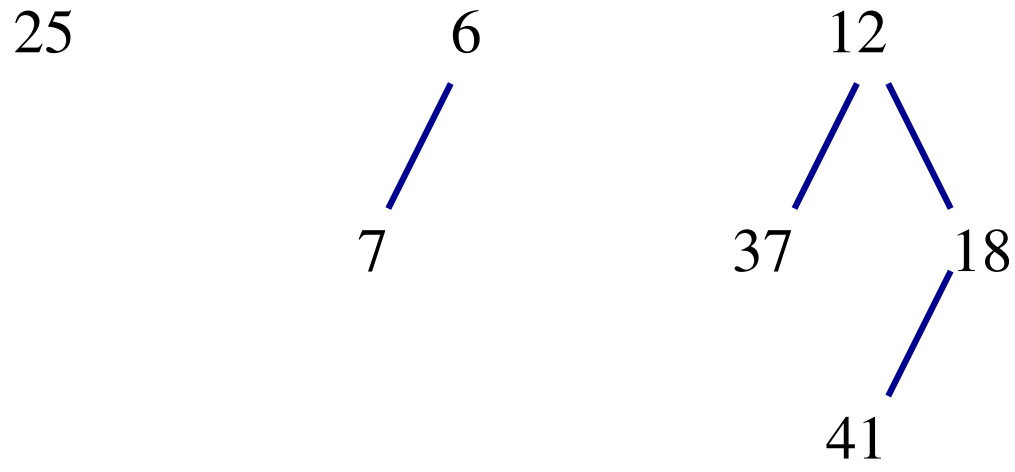
Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x
2. Setze $i \leftarrow 0$
3. **while** es gibt zwei Bäume B_i im Heap **do**
4. Vereinige die beiden B_i so dass die Heap-Ordnung erhalten bleibt
5. Setze $i \leftarrow i + 1$

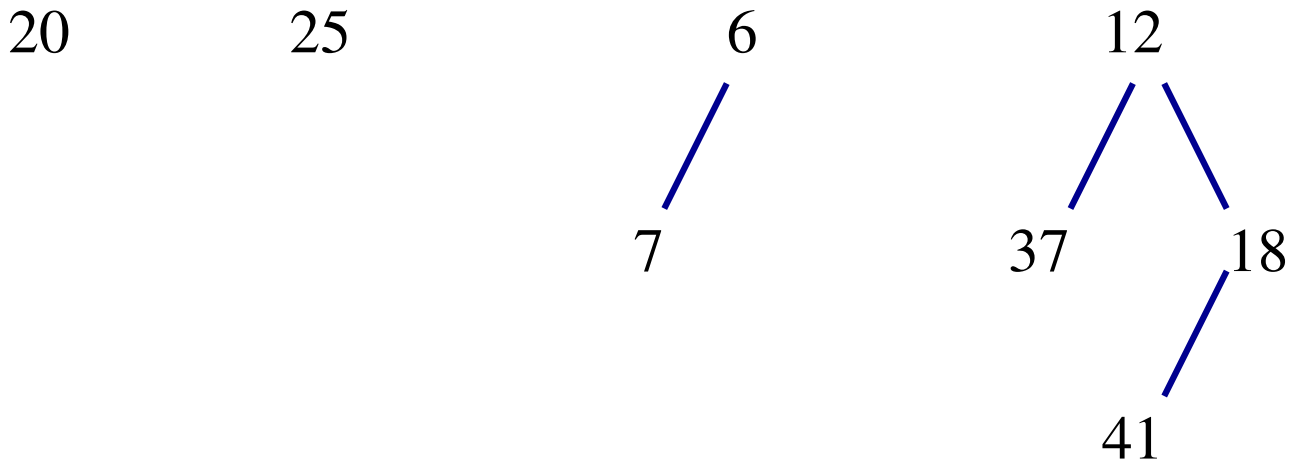
Insert(x) im Binomischen Heap

1. Generiere eine Kopie von B_0 mit dem Schlüssel x
2. Setze $i \leftarrow 0$
3. **while** es gibt zwei Bäume B_i im Heap **do**
4. Vereinige die beiden B_i so dass die Heap-Ordnung erhalten bleibt
5. Setze $i \leftarrow i + 1$

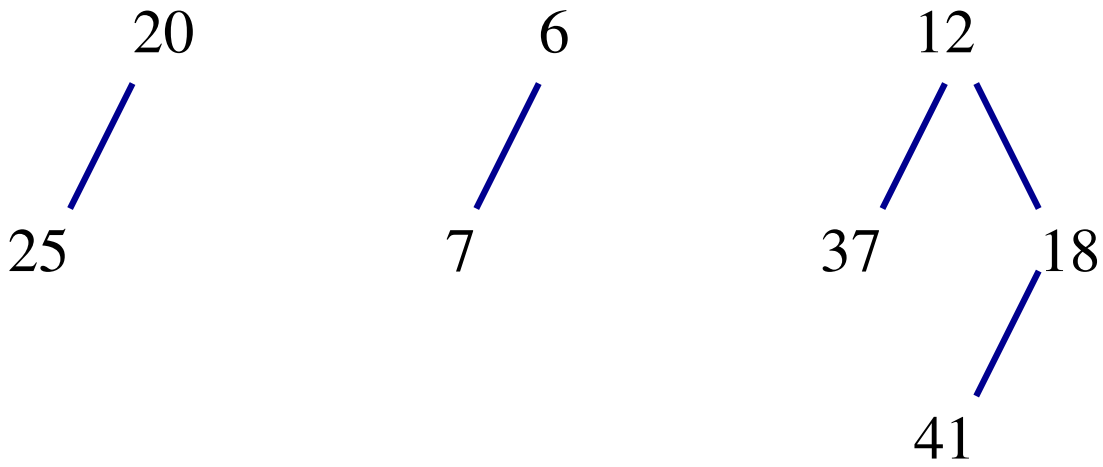
Beispiel: Insert(20)



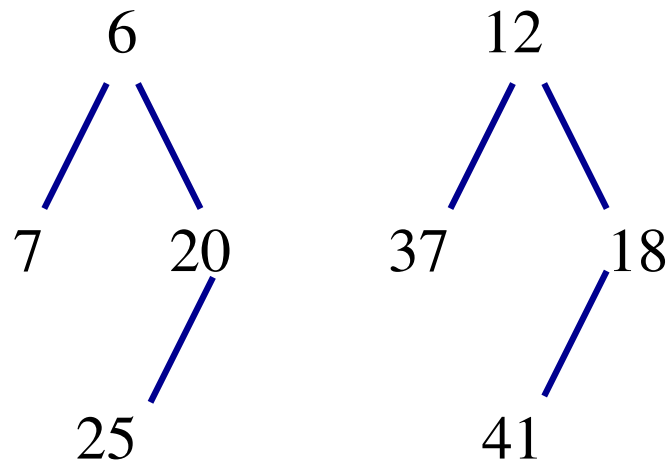
Beispiel: Insert(20)



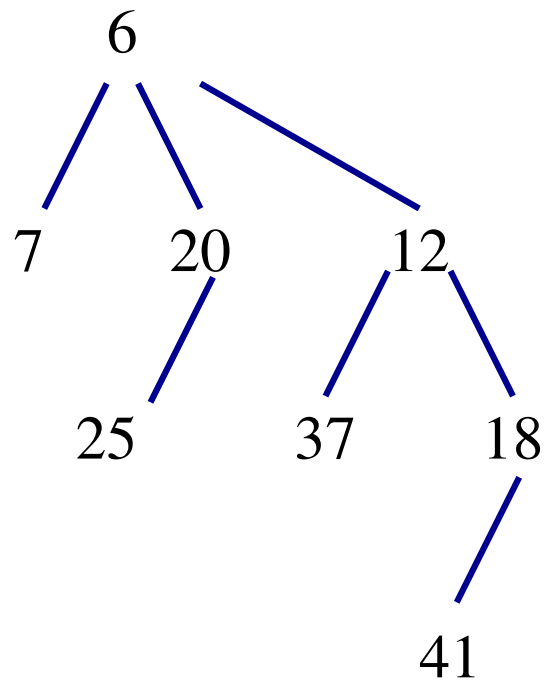
Beispiel: Insert(20)



Beispiel: Insert(20)



Beispiel: Insert(20)



DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w.

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem binomischen Heap H'

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem binomischen Heap H'
6. Sei $M \leftarrow H' \cup H''$

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem binomischen Heap H'
6. Sei $M \leftarrow H' \cup H''$
7. **for** $i = 0, 1, \ldots, \log_2 m$ **do**

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem binomischen Heap H'
6. Sei $M \leftarrow H' \cup H''$
7. **for** $i = 0, 1, \ldots, \log_2 m$ **do**
8. **if** mehr als 2 Bäume B_i in M **then**
9. Vereinige sie so dass die Heap-Ordnung erhalten bleibt

DeleteMin im Binomischen Heap

1. Sei H der Binomische Heap
2. Finde die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' \leftarrow H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem binomischen Heap H'
6. Sei $M \leftarrow H' \cup H''$
7. **for** $i = 0, 1, \ldots, \log_2 m$ **do**
8. **if** mehr als 2 Bäume B_i in M **then**
9. Vereinige sie so dass die Heap-Ordnung erhalten bleibt

Beobachtung

Seien m' Schlüssel in Heap H' sind und m'' Schlüssel in Heap H'' enthalten.

Dann entspricht die **Anzahl der B_i Bäume** in jeder Runde der **Anzahl der 1-Bits** bei der Grundschul-Addition der Zahlen m' und m'' in Binärdarstellung.

Beobachtung

Seien m' Schlüssel in Heap H' sind und m'' Schlüssel in Heap H'' enthalten.

Dann entspricht die **Anzahl der B_i Bäume** in jeder Runde der **Anzahl der 1-Bits** bei der Grundschul-Addition der Zahlen m' und m'' in Binärdarstellung.

Daher gilt:

Folgerung:

In einem binomischen Heap H mit m Schlüsseln dauert eine **DeleteMin Operation** $\mathcal{O}(\log_2 m)$ **Schritte**.

Fibonacci Heaps

Definition:

Ein **Fibonacci Heap** ist eine Menge von Fibonacci-Bäumen so dass

Fibonacci Heaps

Definition:

Ein **Fibonacci Heap** ist eine Menge von Fibonacci-Bäumen so dass

a. jeder Knoten einen Schlüssel speichert,

Fibonacci Heaps

Definition:

Ein **Fibonacci Heap** ist eine Menge von Fibonacci-Bäumen so dass

a. jeder Knoten einen Schlüssel speichert,

b. in jedem Baum die Heap-Ordnung gilt und

Fibonacci Heaps

Definition:

Ein **Fibonacci Heap** ist eine Menge von Fibonacci-Bäumen so dass

a. jeder Knoten einen Schlüssel speichert,

b. in jedem Baum die Heap-Ordnung gilt und

c. ein MIN Zeiger auf die Wurzel mit dem kleinsten Schlüssel zeigt.

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem Fibonacci Heap H'

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem Fibonacci Heap H'
6. Sei $M = H' \cup H''$

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem Fibonacci Heap H'
6. Sei $M = H' \cup H''$
7. **while** es gibt 2 Bäume mit gleichem Wurzelgrad **do**

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem Fibonacci Heap H'
6. Sei $M = H' \cup H''$
7. **while** es gibt 2 Bäume mit gleichem Wurzelgrad **do**
8. Vereinige sie so dass die Heap-Ordnung erhalten bleibt

DeleteMin im Fibonacci Heap

1. Sei H der Fibonacci Heap
2. MIN zeigt die Wurzel w mit dem kleinsten Schlüssel x
3. Sei T_w der binomische Baum mit Wurzel w
4. Sei $H'' = H \setminus T_w$
5. Entferne die Wurzel w. Die verbleibenden Teilbäume (der Kinder von w) entsprechen einem Fibonacci Heap H'
6. Sei $M = H' \cup H''$
7. **while** es gibt 2 Bäume mit gleichem Wurzelgrad **do**
8. Vereinige sie so dass die Heap-Ordnung erhalten bleibt
9. Aktualisiere den MIN Zeiger

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v
4. Prüfe ob MIN auf v_x mit Wert $x - \Delta$ zeigen soll

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v
4. Prüfe ob MIN auf v_x mit Wert $x - \Delta$ zeigen soll
5. **while** v markiert **then**

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v
4. Prüfe ob MIN auf v_x mit Wert $x - \Delta$ zeigen soll
5. **while** v markiert **then**
6. Sei $v' = Vater(v)$. Trenne v von v'

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v
4. Prüfe ob MIN auf v_x mit Wert $x - \Delta$ zeigen soll
5. **while** v markiert **then**
6. Sei $v' = Vater(v)$. Trenne v von v'
7. Prüfe ob MIN auf v zeigen soll

DecreaseKey(wo(x), Δ) im Fibonacci Heap

1. Setze x um Δ herab, sei $v_x = wo(x)$
2. **if** Heap-Ordnung verletzt **then**
3. Sei $v = Vater(v_x)$. Trenne v_x von v
4. Prüfe ob MIN auf v_x mit Wert $x - \Delta$ zeigen soll
5. **while** v markiert **then**
6. Sei $v' = Vater(v)$. Trenne v von v'
7. Prüfe ob MIN auf v zeigen soll
8. Setze $v = Vater(v)$ (rekursiv wiederholt nach oben)
9. Markiere v (tiefster unmarkierter Vorfahre von v_x)

Splay-Bäume

Definition:

Ein **Splay-Baum** ist ein binärer Suchbaum.

Für einen Knoten mit Schlüssel x sind im linken (rechten) Teilbaum genau die Knoten mit Schlüssel kleiner (größer) als x.

Splay-Bäume

Definition:

Ein **Splay-Baum** ist ein binärer Suchbaum.

Für einen Knoten mit Schlüssel x sind im linken (rechten) Teilbaum genau die Knoten mit Schlüssel kleiner (größer) als x.

Splay(x) Operation:

- Starte an der Wurzel
- Suche Knoten v mit **größtem Schlüssel** $\leq x$.
- Bringe v durch Links- und Rechtsrotationen an die Wurzelposition.

Operationen

Für alle Operationen **führe immer zuerst Splay(x) aus**.

Danach gehen wir wie folgt vor:

Lookup(x): Wurzel ist der Knoten mit Schlüssel x, falls vorhanden.

Operationen

Für alle Operationen **führe immer zuerst Splay(x) aus**.

Danach gehen wir wie folgt vor:

Lookup(x): Wurzel ist der Knoten mit Schlüssel x, falls vorhanden.

Insert(x): Falls Schlüssel x an der Wurzel: Stopp. Sonst:

Sei T_L der linke und T_R der rechte Teilbaum der Wurzel. Erstelle neue Wurzel mit x. Hänge bisherige Wurzel und T_L links an, hänge T_R rechts an.

Operationen

Für alle Operationen **führe immer zuerst Splay(x) aus**.

Danach gehen wir wie folgt vor:

Lookup(x): Wurzel ist der Knoten mit Schlüssel x, falls vorhanden.

Insert(x): Falls Schlüssel x an der Wurzel: Stopp. Sonst:

Sei T_L der linke und T_R der rechte Teilbaum der Wurzel. Erstelle neue Wurzel mit x. Hänge bisherige Wurzel und T_L links an, hänge T_R rechts an.

Remove(x): Falls Schlüssel $\neq x$ an der Wurzel: Stopp. Sonst:

Entferne Wurzel. Sei T_L der linke und T_R der rechte Teilbaum. Führe Splay(x) auf linkem Teilbaum aus. Hänge T_R rechts an neue Wurzel von T_L.

Operationen

Für alle Operationen **führe immer zuerst Splay(x) aus**.

Danach gehen wir wie folgt vor:

Lookup(x): Wurzel ist der Knoten mit Schlüssel x, falls vorhanden.

Insert(x): Falls Schlüssel x an der Wurzel: Stopp. Sonst:

Sei T_L der linke und T_R der rechte Teilbaum der Wurzel. Erstelle neue Wurzel mit x. Hänge bisherige Wurzel und T_L links an, hänge T_R rechts an.

Remove(x): Falls Schlüssel $\neq x$ an der Wurzel: Stopp. Sonst:

Entferne Wurzel. Sei T_L der linke und T_R der rechte Teilbaum. Führe Splay(x) auf linkem Teilbaum aus. Hänge T_R rechts an neue Wurzel von T_L.

Bis auf konstanten Faktor gilt:

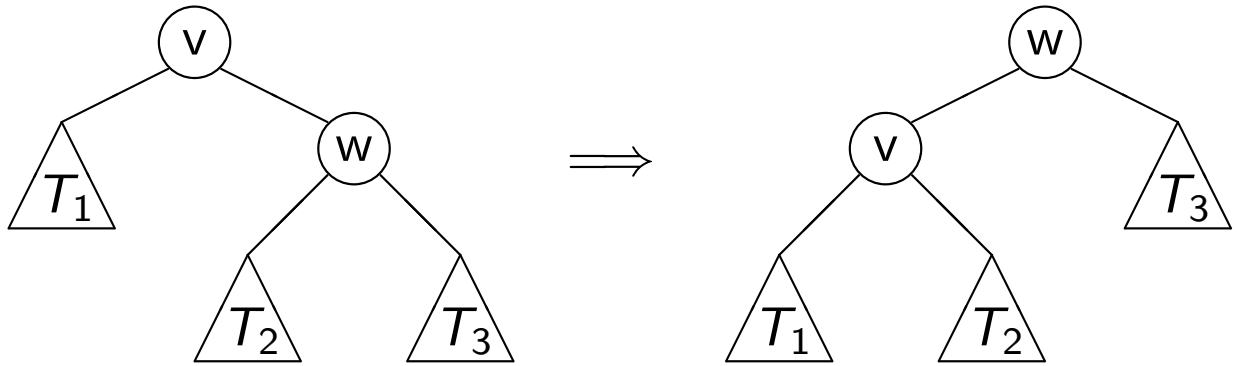
Aufwand für **jede Operation** $\equiv$ Aufwand für **eine Splay-Operation**

Splay-Operation: Rotationen

Linksrotation bei v: Rechtes Kind w ersetzt Elternknoten v. v wird linkes Kind von w.

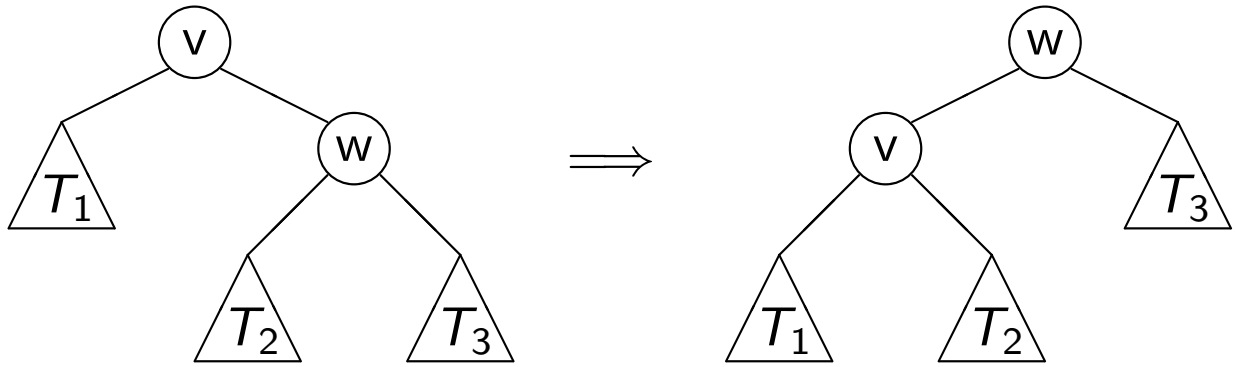
Splay-Operation: Rotationen

Linksrotation bei v: Rechtes Kind w ersetzt Elternknoten v. v wird linkes Kind von w.

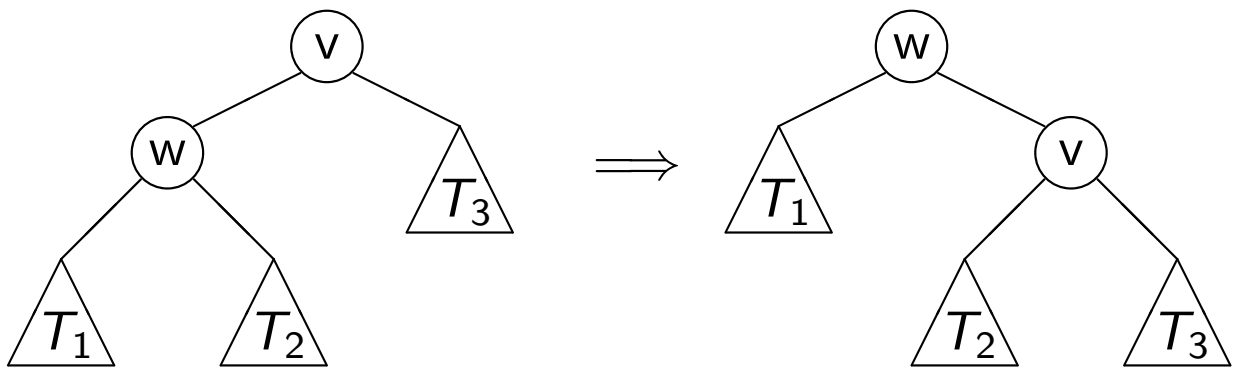


Splay-Operation: Rotationen

Linksrotation bei v: Rechtes Kind w ersetzt Elternknoten v. v wird linkes Kind von w.



Rechtsrotation bei v: Entsprechend.



Splay-Operation: Rotationen

Ziel: Bringe Knoten y auf die Höhe seines Vaters oder Großvaters.

Je nach Lage des **Knotens** y, seines **Vaters** v und seines **Großvaters** g brauchen wir verschiedene Rotationen:

Zick-Zick: y, v beide linke Kinder – Rechtsrotationen bei g, dann bei v
y, v beide rechte Kinder – Linksrotationen bei g, dann bei v

Splay-Operation: Rotationen

Ziel: Bringe Knoten y auf die Höhe seines Vaters oder Großvaters.

Je nach Lage des **Knotens** y, seines **Vaters** v und seines **Großvaters** g brauchen wir verschiedene Rotationen:

Zick-Zick: y, v beide linke Kinder – Rechtsrotationen bei g, dann bei v
y, v beide rechte Kinder – Linksrotationen bei g, dann bei v

Zick-Zack: y li., v re. Kind – Rechtsrotation bei v, dann Links bei g
y re., v li. Kind – Linksrotation bei v, dann Rechts bei g

Splay-Operation: Rotationen

Ziel: Bringe Knoten y auf die Höhe seines Vaters oder Großvaters.

Je nach Lage des **Knotens** y, seines **Vaters** v und seines **Großvaters** g brauchen wir verschiedene Rotationen:

Zick-Zick: y, v beide linke Kinder – Rechtsrotationen bei g, dann bei v
y, v beide rechte Kinder – Linksrotationen bei g, dann bei v

Zick-Zack: y li., v re. Kind – Rechtsrotation bei v, dann Links bei g
y re., v li. Kind – Linksrotation bei v, dann Rechts bei g

Zick: y linkes Kind, v Wurzel – Rechtsrotation bei v
y rechtes Kind, v Wurzel – Linksrotation bei v

Potenzial und Laufzeit

Definition:

Sei $|T_v|$ die **Anzahl Knoten im Teilbaum** mit Wurzel v. Wir nennen

$$R_T(v) = \log_2(|T_v|)$$

den **Rang von Knoten** v **in** T und

$$\Phi(T) = \sum_{v \in V} R_T(v) = \sum_{v \in V} \log_2(|T_v|)$$

die Potenzialfunktion.

Potenzial und Laufzeit

Definition:

Sei $|T_v|$ die **Anzahl Knoten im Teilbaum** mit Wurzel v. Wir nennen

$$R_T(v) = \log_2(|T_v|)$$

den **Rang von Knoten** v **in** T und

$$\Phi(T) = \sum_{v \in V} R_T(v) = \sum_{v \in V} \log_2(|T_v|)$$

die Potenzialfunktion.

Theorem:

Bei anfänglich leerem Suchbaum hat jede Folge von n Operationen des Typs Insert, Remove und Lookup eine **Gesamtlaufzeit von** $O(n \log n)$. Die **amortisierte Laufzeit jeder Operation** ist $O(\log n)$.

Dynamic Optimality Conjecture

- Gegeben ein binärer Suchbaum T mit n Knoten
- Sequenz σ von m Lookup Operationen

Dynamic Optimality Conjecture

- Gegeben ein binärer Suchbaum T mit n Knoten
- Sequenz σ von m Lookup Operationen

- Algorithmus A kann in Runde i eine Anzahl r_i Rotationen ausführen.
- Anzahl inspizierter Knoten auf dem Suchpfades in Runde i sei ℓ_i.
- $\text{Kosten}_A(T, \sigma) = \sum_{i=1}^{m}(r_i + \ell_i)$

Dynamic Optimality Conjecture

- Gegeben ein binärer Suchbaum T mit n Knoten
- Sequenz σ von m Lookup Operationen
- Algorithmus A kann in Runde i eine Anzahl r_i Rotationen ausführen.
- Anzahl inspizierter Knoten auf dem Suchpfades in Runde i sei ℓ_i.
- $\text{Kosten}_A(T, \sigma) = \sum_{i=1}^{m}(r_i + \ell_i)$
- Splay-Baum: Sucht angefragtes Element und rotiert entlang des Suchpfades zur Wurzel, also immer $r_i + 1 \leq \ell_i \leq r_i + 2$.

Dynamic Optimality Conjecture

- Gegeben ein binärer Suchbaum T mit n Knoten
- Sequenz σ von m Lookup Operationen

- Algorithmus A kann in Runde i eine Anzahl r_i Rotationen ausführen.
- Anzahl inspizierter Knoten auf dem Suchpfades in Runde i sei ℓ_i.
- $\text{Kosten}_A(T, \sigma) = \sum_{i=1}^{m}(r_i + \ell_i)$

- Splay-Baum: Sucht angefragtes Element und rotiert entlang des Suchpfades zur Wurzel, also immer $r_i + 1 \leq \ell_i \leq r_i + 2$.
- Offline-Algorithmus: Optimiert mit Wissen über σ macht evtl. weniger Rotationen, z.B. wenn Element später nicht mehr angefragt

Dynamic Optimality Conjecture

- Gegeben ein binärer Suchbaum T mit n Knoten
- Sequenz σ von m Lookup Operationen

- Algorithmus A kann in Runde i eine Anzahl r_i Rotationen ausführen.
- Anzahl inspizierter Knoten auf dem Suchpfades in Runde i sei ℓ_i.
- $\text{Kosten}_A(T, \sigma) = \sum_{i=1}^{m}(r_i + \ell_i)$

- Splay-Baum: Sucht angefragtes Element und rotiert entlang des Suchpfades zur Wurzel, also immer $r_i + 1 \leq \ell_i \leq r_i + 2$.
- Offline-Algorithmus: Optimiert mit Wissen über σ macht evtl. weniger Rotationen, z.B. wenn Element später nicht mehr angefragt

Korollar:

Splay-Bäume haben Wettbewerbsfaktor $O(\log n)$.

Dynamic Optimality Conjecture:
Splay-Bäume haben Wettbewerbsfaktor $O(1)$.