ETH zürich

Übung 4

Datenstrukturen und Algorithmen, D-MATH, ETH Zurich

Programm von heute

Feedback letzte Übung

Wiederholung Theorie
- Amortisierte Analyse
- Skiplisten

Zur Bonusaufgabe

Code-Beispiel: Dynamischer Vektor

1. Feedback letzte Übung

Eier werfen

- Strategie für beliebig viele Eier?

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?
  - Von unten anfangen. n Versuche.

Eier werfen

Strategie mit zwei Eiern

- 1. Ansatz. Intervalle gleicher Länge: Unterteile n in k Intervalle. Maximale Anzahl Versuche:

Eier werfen

Strategie mit zwei Eiern

- 1. Ansatz. Intervalle gleicher Länge: Unterteile n in k Intervalle.
  Maximale Anzahl Versuche: $f(k) = k + n/k - 1$
  Minimiere maximale Anzahl Versuche:

Eier werfen

Strategie mit zwei Eiern

- 1. Ansatz. Intervalle gleicher Länge: Unterteile n in k Intervalle.
  Maximale Anzahl Versuche: $f(k) = k + n/k - 1$
  Minimiere maximale Anzahl Versuche: $f'(k) = 1 - n/k^2 = 0 \Rightarrow k = \sqrt{n}$.
  $n = 100 \Rightarrow 19$ Versuche. $\Theta(\sqrt{n})$
- Zweiter Ansatz: Beziehe ersten Wurfversuch in die Berechnung ein mit kleiner werdenden Intervallen. Wähle kleinstes s mit
  $s + s - 1 + s - 2 + ... + 1 = s(s+1)/2 \geq 100 \Rightarrow s = 14$. Maximale Anzahl Versuche: $s \in \Theta(\sqrt{n})$

Asymptotisch sind beide Methoden gleich gut. Praktisch ist der zweite Ansatz vorzuziehen.

Selection-Algorithmus

- Was passiert bei vielen gleichen Elementen?
- $99, 99, \ldots, 99$, Pivot 99, kleiner Partition leer, grösser Parition hat $n - 1$ mal 99.
- Kann Laufzeit auf n^2 verschlechtern
- Lösung?

Selection-Algorithmus

- Bei Gleichheit mit Pivot, wechsle Partition ab.

Selection-Algorithmus

- Bei Gleichheit mit Pivot, wechsle Partition ab.
- Erweitere Algorithmus um explizit Anzahl gleicher Elemente zu behandeln.

2.1 Amortisierte Analyse

Amortisierte Laufzeitanalyse

Drei Methoden

- Aggregierte Analyse
- Kontomethode
- Potentialmethode

Beispiel: einfaches Wörterbuch

Unterstützt Operationen Insert und Find. Idee:

- Familie von Arrays A_i mit Grösse 2^i
- Jedes Array ist entweder ganz leer oder ganz voll und speichert seine Elemente in sortierter Reihenfolge
- Zwischen den Arrays besteht keine weitere Beziehung

Daten $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0&: \quad [50] \\
A_1&: \quad [8, 99] \\
A_2&: \quad \emptyset \\
A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

Beispiel: einfaches Wörterbuch

Daten $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0&: \quad [50] \\
A_1&: \quad [8, 99] \\
A_2&: \quad \emptyset \\
A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

Algorithmus **`Find`**:

Beispiel: einfaches Wörterbuch

Daten $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0&: \quad [50] \\
A_1&: \quad [8, 99] \\
A_2&: \quad \emptyset \\
A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

Algorithmus **`Find`**: Durchlaufen aller Arrays, jeweils binäre Suche
Worst-case Laufzeit :

Beispiel: einfaches Wörterbuch

Daten $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0&: \quad [50] \\
A_1&: \quad [8, 99] \\
A_2&: \quad \emptyset \\
A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

Algorithmus **`Find`**: Durchlaufen aller Arrays, jeweils binäre Suche
Worst-case Laufzeit : $\Theta(\log^2 n)$,

$$\log 1 + \log 2 + \log 4 + \cdots + \log 2^k = \sum_{i=0}^{k} \log_2 2^i = \frac{k \cdot (k+1)}{2} \in \Theta(\log^2 n).$$

$(k = \lfloor \log_2 n \rfloor)$

Beispiel: einfaches Wörterbuch

Algorithmus **`Insert(x)`**:

Beispiel: einfaches Wörterbuch

Algorithmus **`Insert(x)`**:

- Neues Array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- Solange $A_i \neq \emptyset$, setze $A'_{i+1} = \text{Merge}(A_i, A'_i)$, $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Setze $A_i \leftarrow A'_i$

Insert(11)

$$
\begin{array}{ll} A_0: & [50] \\ A_1: & [8, 99] \\ A_2: & \emptyset \\ A_3: & [1, 10, 18, \ldots, 75] \end{array}
\qquad
\begin{array}{ll} A'_0: & [11] \\ A'_1: & [11, 50] \\ A'_2: & [8, 11, 50, 99] \end{array}
\qquad \Rightarrow \qquad
\begin{array}{ll} A_0: & \emptyset \\ A_1: & \emptyset \\ A_2: & [8, 11, 50, 99] \\ A_3: & [1, 10, 18, \ldots, 75] \end{array}
$$

Kosten Insert

Im Folgenden: $n = 2^k$, $k = \log_2 n$

Annahme: Erzeugen neues Array A'_i der Länge 2^i (und, für $i > 0$ anschliessendes Zusammenführen von A'_{i-1} und A_{i-1}) hat Kosten $\Theta(2^i)$

Im schlechtesten Fall erzeugt das Einfügen eines Elements $\log_2 n$ solche Operationen. $\Rightarrow$ **Worst-case Kosten Insert**:

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \in \Theta(n).$$

Aggregatanalyse

Level	Kosten	Beispiel Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beobachtung: startet man mit einem leeren Container, so gelangt die Einfügesequenz jedes Mal auf Level 0, jedes zweite Mal auf Level 1 (mit Kosten 2), jedes vierte Mal das Level 2 (mit Kosten 4), jedes achte Mal das Level 3 (mit Kosten 8) etc.

Aggregatanalyse

Level	Kosten	Beispiel Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Beobachtung: startet man mit einem leeren Container, so gelangt die Einfügesequenz jedes Mal auf Level 0, jedes zweite Mal auf Level 1 (mit Kosten 2), jedes vierte Mal das Level 2 (mit Kosten 4), jedes achte Mal das Level 3 (mit Kosten 8) etc.

Gesamtkosten: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} = (k+1)n \in \Theta(n \log n)$.
Amortisierte Kosten pro Operation: $\Theta((n \log n)/n) = \Theta(\log n)$.

Kontomethode

- Jedes Element i $(1 \leq i \leq n)$ zahlt $a_i = \log_2 n$ Münzen, wenn es in die Datenstruktur eingefügt wird.
- Damit bezahlt das Element das Anlegen des ersten Arrays und jeden weiteren Merge-Schritt, welcher auftreten kann, bis das Element im Array A_k angekommen ist.
- Das Konto weist damit immer genügend Kredit auf, um alle Merge-Operationen zu bezahlen.

$\Rightarrow$ **Amortisierte Kosten** für das Einfügen $\mathcal{O}(\log n)$

Potentialmethode

Wir wissen von der Kontomethode, dass jedes Element auf dem Weg nach oben $\log n$ Münzen benötigt, d.h. dass ein **Element auf dem Level** i **noch** $k - i$ **Münzen** haben sollte. Wir verwenden das **Potential**

$$\Phi_j = \sum_{0 \leq i \leq k : A_i \neq \emptyset} (k - i) \cdot 2^i$$

Potentialmethode

Für die **Änderung des Potentials** $\Phi_j - \Phi_{j-1}$ müssen wir nur die unteren l Levels betrachten, welche zum Zeitpunkt $j-1$ belegt sind (Analogie zum binären Zähler). Sei also l der kleinste Index, so dass A_l leer ist.

Nach dem Zusamenführen der Arrays $A_0 \dots A_{l-1}$ sind Arrays $A_i, 0 \leq i < l$ leer und das Level A_l ist nun belegt. Also:

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Reale Kosten:

$$t_j = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$$

Potentialmethode

$$
\begin{aligned}
\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + l\cdot 2^l - 2^{l+1} + 2 \\
&= k - 2^{l+1} + 2 \\
\Phi_j - \Phi_{j-1} + t_j &= k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)
\end{aligned}
$$

$\sum i \cdot \lambda^i$

Immer der gleiche Trick:

$$\begin{aligned}
\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i &= \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i \\
&= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i \\
&= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1 \\
(\lambda - 1) \cdot \sum_{i=0}^{n} i \cdot \lambda^i &= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1
\end{aligned}$$

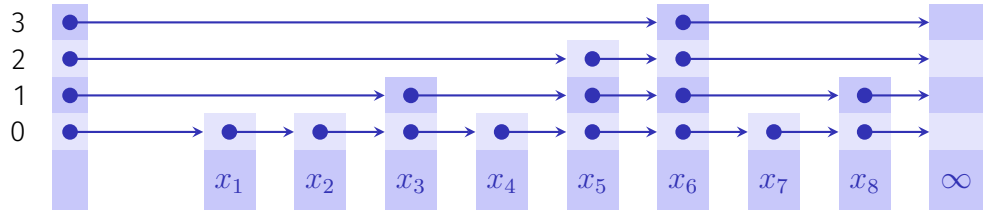
Für $\lambda = 2$:

$$\sum_{i=0}^{n} i \cdot 2^i = n \cdot 2^{n+1} - 2^{n+1} + 1 + 1 = (n-1) \cdot 2^{n+1} + 2$$

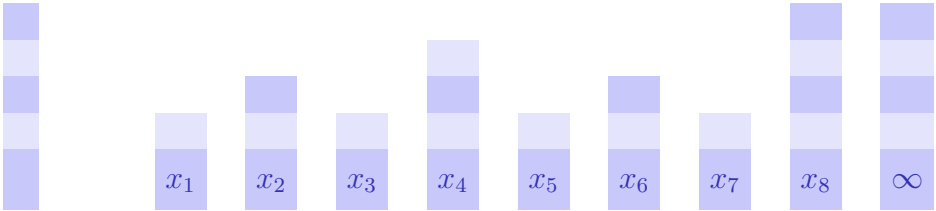
2.2 Skiplisten

Randomisierte Skipliste

Idee: Füge jeweils einen Knoten mit zufälliger Höhe H ein, wobei $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.

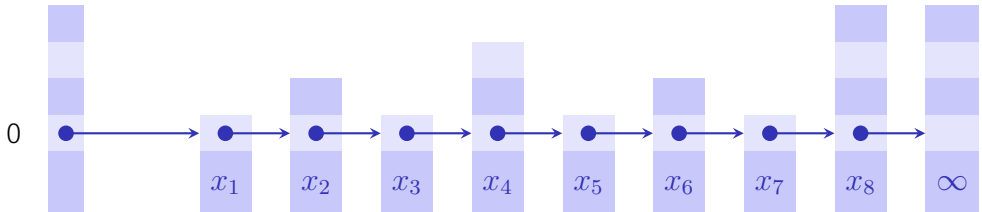


Randomisierte Skipliste: Element finden



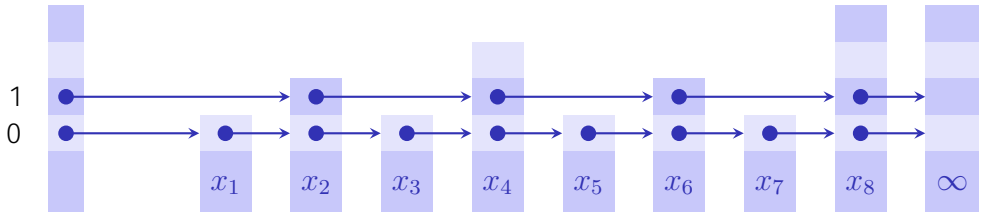
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



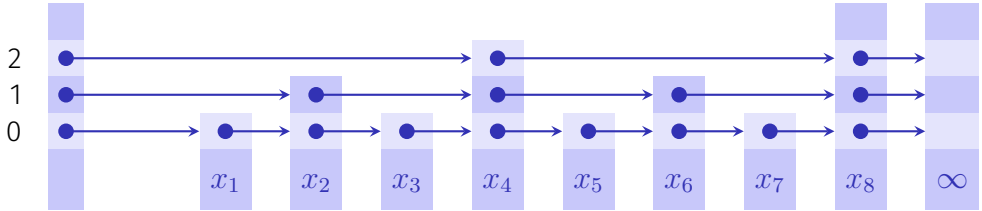
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



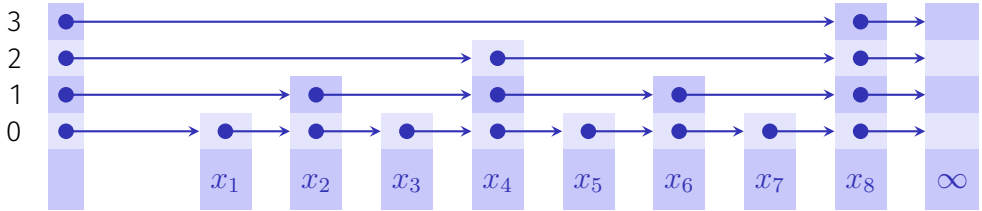
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

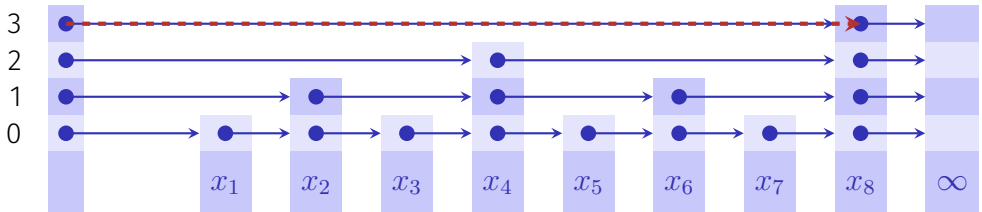
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

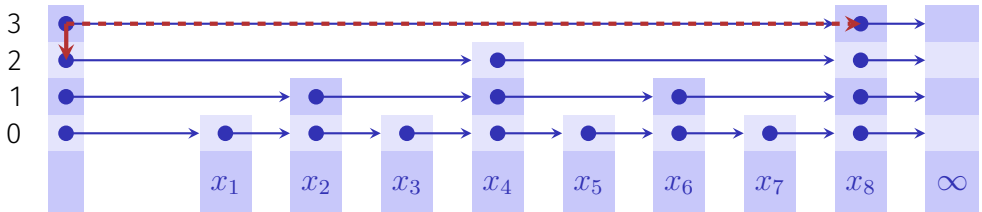
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

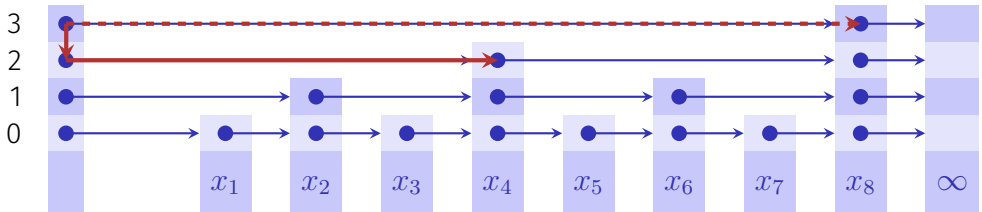
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

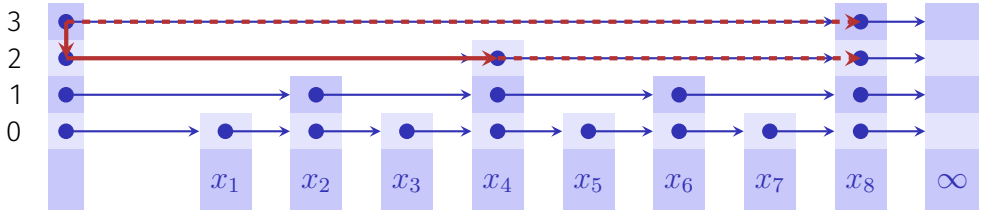
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

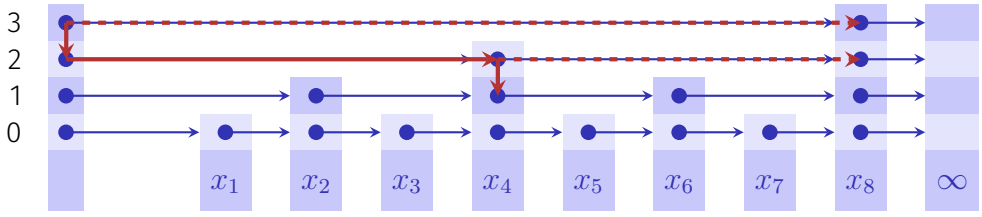
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

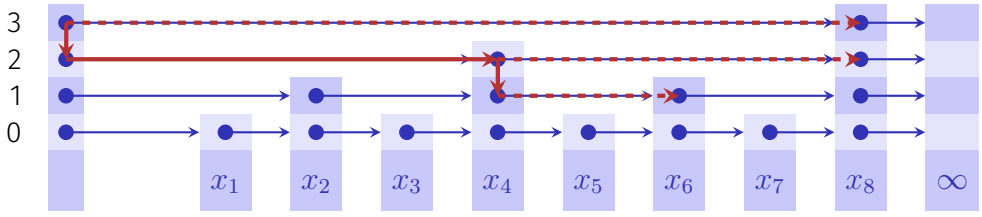
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

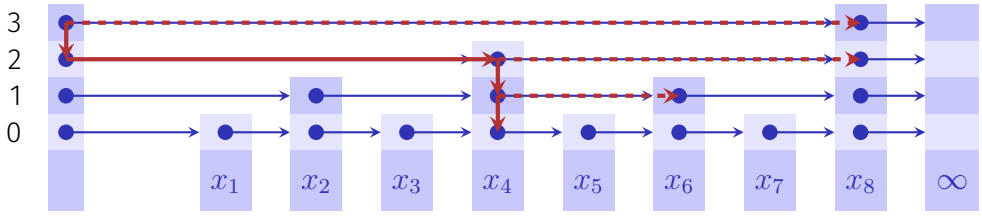
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

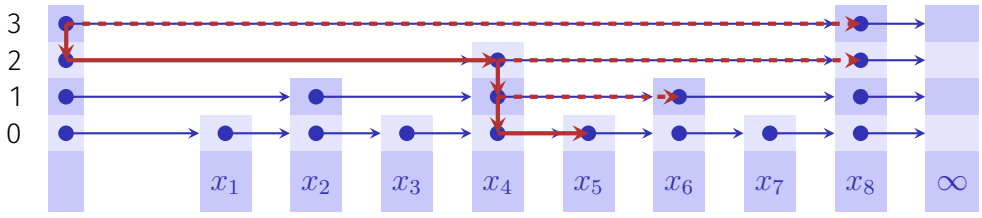
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

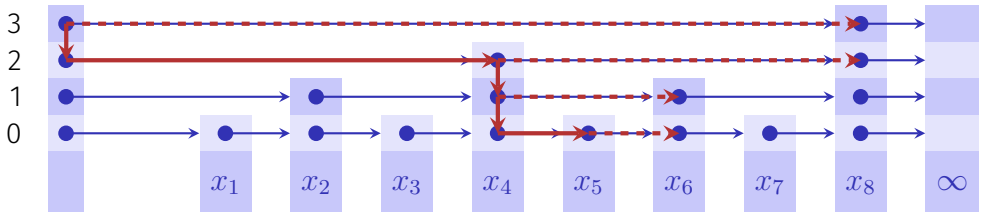
Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

Randomisierte Skipliste: Element finden



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Beispiel: Suche nach einem Schlüssel x mit $x_5 < x < x_6$.

Skiplisten-Interface

```
template<typename T> class SkipList {
public:
  SkipList();
  ~SkipList();

  void insert(const T& value);
  void erase(const T& value);

  // iterator implementation ...
};
```

Teilweise implementiert:

- Eine Klasse `Node` speichert ein Element `value` vom Typ `T` und einen `std::vector` (`forward`) mit Pointern auf nachfolgende Nodes.
- Erste Node (ohne Wert): `head`.
- `forward[0]` zeigt auf die jeweils nächste Node in der Liste.
- Wir verwenden das in einem bereits implementierten Iterator.

3. Zur Bonusaufgabe

Implementiere `insert` und `erase`

`insert(const T& value)`

- erstelle neue Node
- wähle zufällige Anzahl von Leveln
- finde, für jeden Level, die nächst-kleinere Node
- setze Pointer von den vorherigen Nodes und der neuen Node

Implementiere `insert` und `erase`

`insert(const T& value)`

- erstelle neue Node
- wähle zufällige Anzahl von Leveln
- finde, für jeden Level, die nächst-kleinere Node
- setze Pointer von den vorherigen Nodes und der neuen Node

`erase(const T& value)`

- finde erst kleinere Node
- überprüfe ob nächste Node den Wert `value` hat
- Pointer entsprechend setzen
- gegebnenfalls Node löschen

Implementiere `insert` und `erase`

`insert(const T& value)`

- erstelle neue Node
- wähle zufällige Anzahl von Leveln
- finde, für jeden Level, die nächst-kleinere Node
- setze Pointer von den vorherigen Nodes und der neuen Node

`erase(const T& value)`

- finde erst kleinere Node
- überprüfe ob nächste Node den Wert `value` hat
- Pointer entsprechend setzen
- gegebnenfalls Node löschen

Warnung: Es können gleiche Werte mehrfach vorkommen.

Wiederholung dynamisch alloziierter Speicher

Sehr wichtig: Jedes `new` braucht sein `delete` und nur eins!

Wiederholung dynamisch alloziierter Speicher

Sehr wichtig: Jedes `new` braucht sein `delete` und nur eins!

Deshalb “Rule of three”:

- constructor
- copy constructor
- destructor

Wiederholung dynamisch alloziierter Speicher

Sehr wichtig: Jedes `new` braucht sein `delete` und nur eins!

Deshalb “Rule of three”:

- constructor
- copy constructor
- destructor

Faule Variante: „Rule of two”:

- niemals kopieren (unsicher)
- mache copy constructor privat (sicher) oder deleted

4. Code-Beispiel: Dynamischer Vektor

Vorbereitung für Deque-Aufgabe

Fragen?