Datenstrukturen und Algorithmen

Übung 4

FS 2018

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
   - Selbstanordnung
   - Skip Lists
3. Programmieraufgabe

Heapsort-Struktur

- Welche Funktionen um Heapsort zu implementieren?

Heapsort-Struktur

```
void sink(...);
void heapify(...);
void heapsort(...);
```

- heapify geht auch inline
- Signatur der Funktionen (für std::vector)?

Heapsort-Struktur

```
void sink(vector<int>& A, size_t index, size_t size
void heapify(vector<int>& A);
void heapsort(vector<int>& A);
```

- Generisch (z.B. für MyVector)?

Heapsort-Struktur

```
template <typename X>
void sink(X& A, size_t index, size_t size);

template <typename X>
void heapify(X& A);

template <typename X>
void heapsort(X& A);
```

2. Wiederholung Theorie

Amortisierte Laufzeitanalyse

Bezeichne t_i die realen Kosten der Operation i.

Potentialfunktion $\Phi_i \geq 0$ für den "Kontostand" nach i Operationen.

Amortisierte Kosten der i-ten Operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

Es gilt

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

Ziel: Suche Potentialfunktion, die teure Operationen ausgleicht.

Selbstanordnung

Problematisch bei der Verwendung verketteter Listen: lineare Suchzeit

Idee: Versuche, die Listenelemente so anzuordnen, dass Zugriffe über die Zeit hinweg schneller möglich sind

Zum Beispiel

- Transpose: Bei jedem Zugriff auf einen Schlüssel wird dieser um eine Position nach vorne bewegt.
- Move-to-Front (MTF): Bei jedem Zugriff auf einen Schlüssel wird dieser ganz nach vorne bewegt.

Transpose

Transpose:

Worst case: n Wechselnde Zugriffe auf k_{n-1} und k_n.

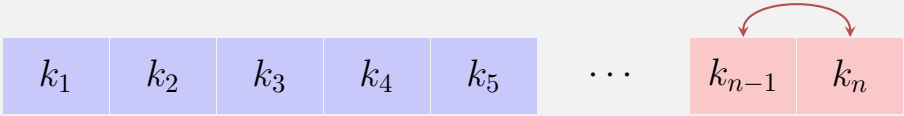
Transpose

Transpose:

Worst case: n Wechselnde Zugriffe auf k_{n-1} und k_n.

Transpose

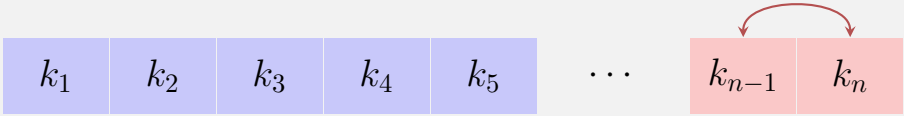
Transpose:



Worst case: n Wechselnde Zugriffe auf k_{n-1} und k_n.

Transpose

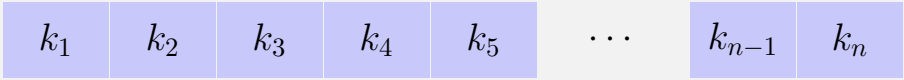
Transpose:



Worst case: n Wechselnde Zugriffe auf k_{n-1} und k_n. Laufzeit: $\Theta(n^2)$

Move-to-Front

Move-to-Front:



n wechselnde Zugriffe auf k_{n-1} und k_n.

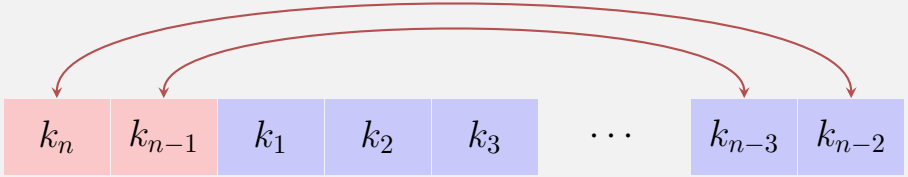
Move-to-Front

Move-to-Front:

n wechselnde Zugriffe auf k_{n-1} und k_n.

Move-to-Front

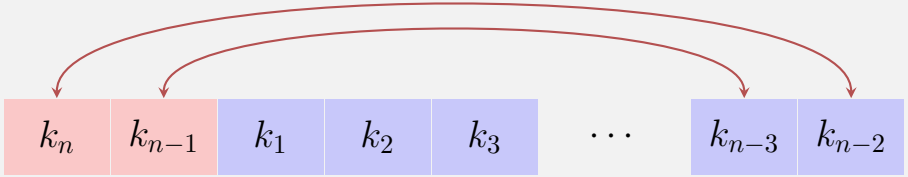
Move-to-Front:



n wechselnde Zugriffe auf k_{n-1} und k_n.

Move-to-Front

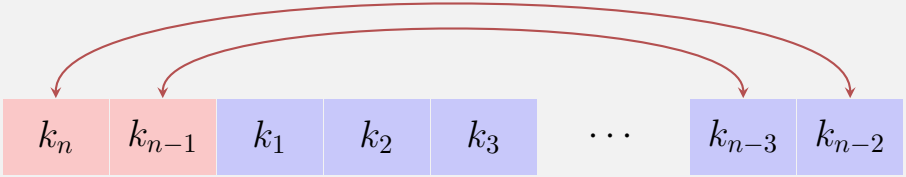
Move-to-Front:



n wechselnde Zugriffe auf k_{n-1} und k_n. Laufzeit: $\Theta(n)$

Move-to-Front

Move-to-Front:

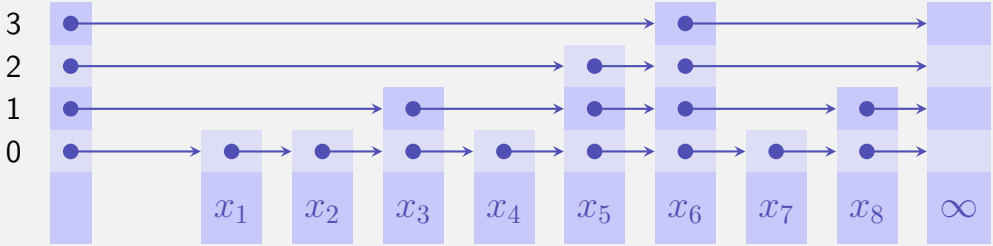


n wechselnde Zugriffe auf k_{n-1} und k_n. Laufzeit: $\Theta(n)$

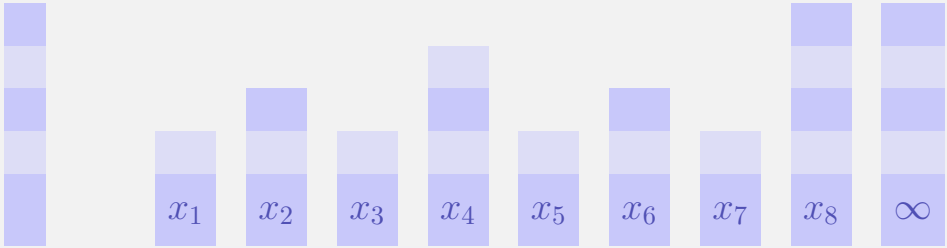
Man kann auch hier Folge mit quadratischer Laufzeit angeben, z.B. immer das letzte Element. Aber dafür ist keine offensichtliche Strategie bekannt, die viel besser sein könnte als MTF.

Randomisierte Skipliste

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

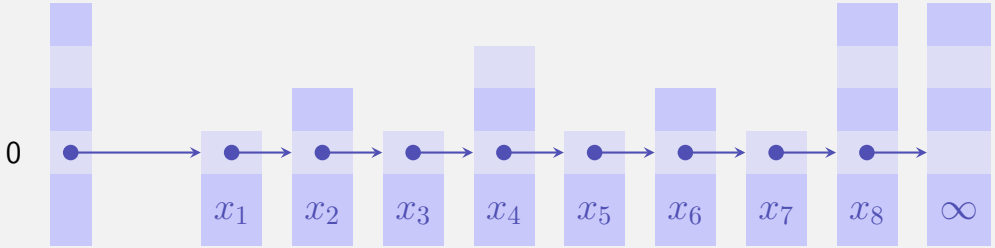


Randomisierte Skipliste: Element finden



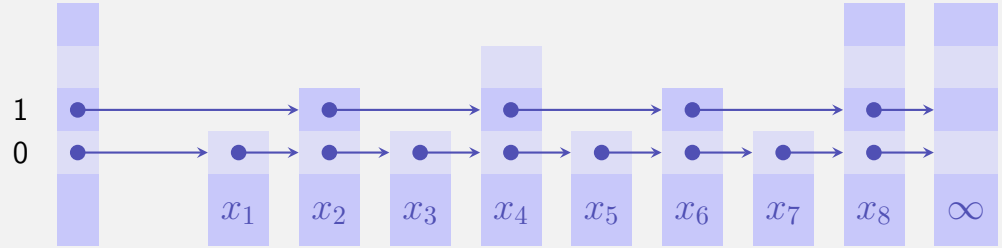
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



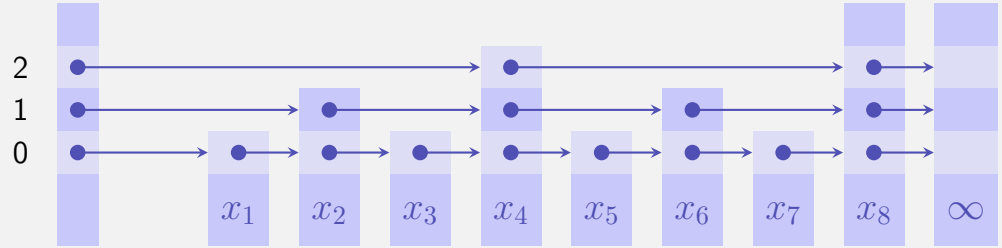
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



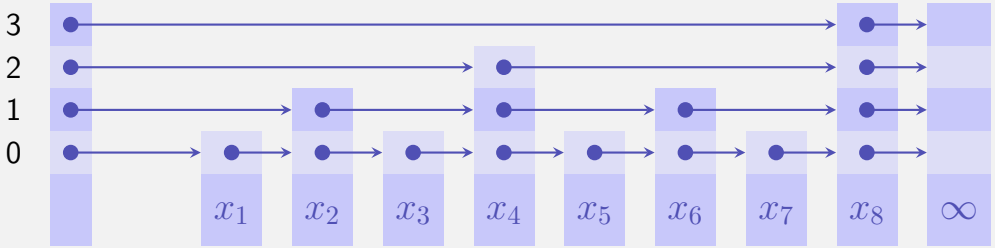
$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$

Randomisierte Skipliste: Element finden



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

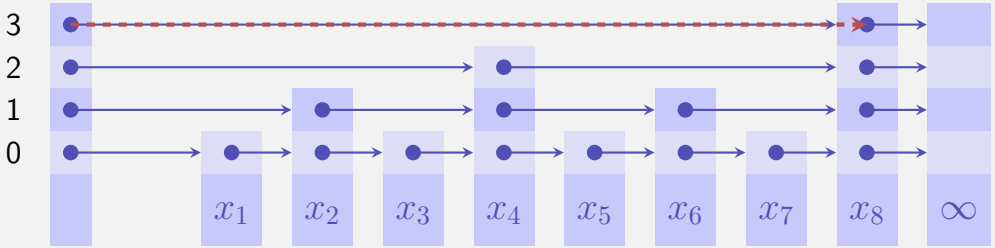
Randomisierte Skipliste: Element finden



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

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

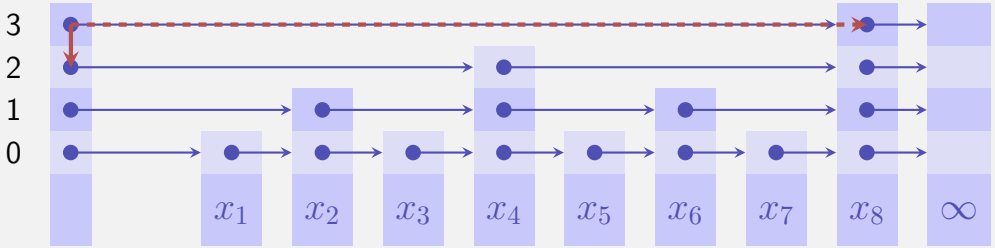
Randomisierte Skipliste: Element finden



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

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

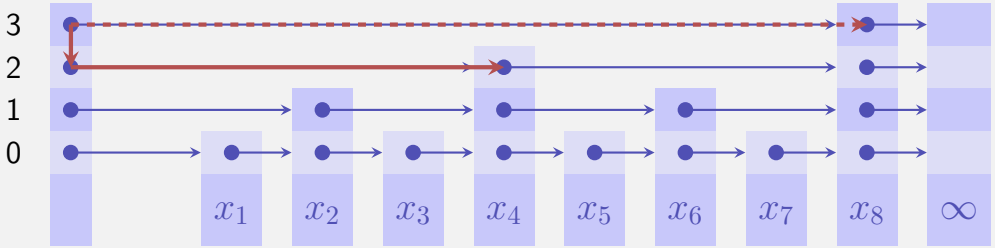
Randomisierte Skipliste: Element finden



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

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

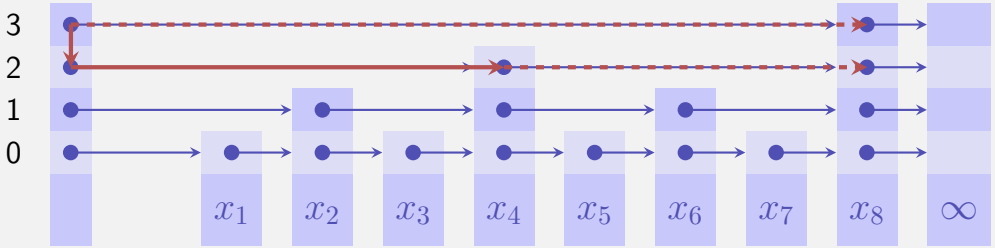
Randomisierte Skipliste: Element finden



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

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

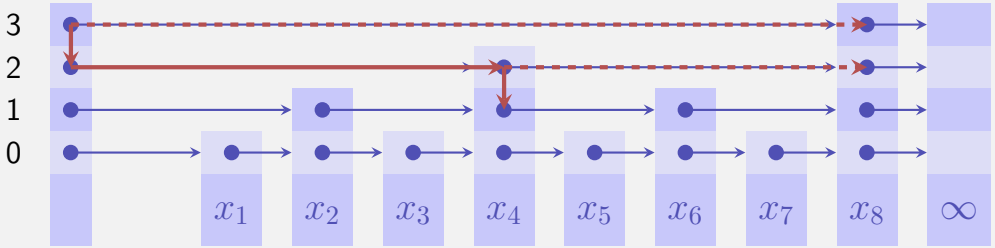
Randomisierte Skipliste: Element finden



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

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

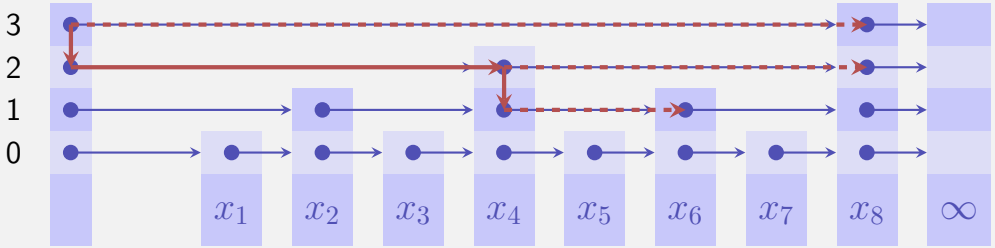
Randomisierte Skipliste: Element finden



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

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

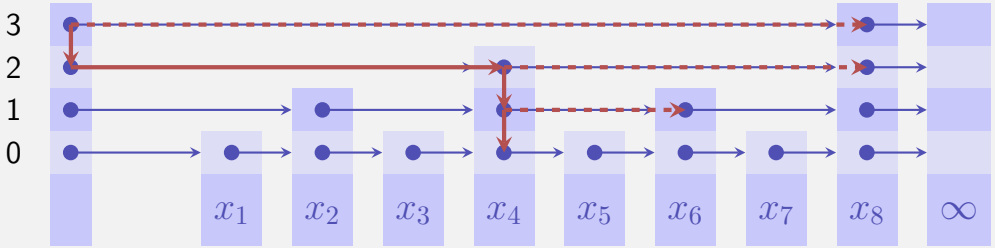
Randomisierte Skipliste: Element finden



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

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

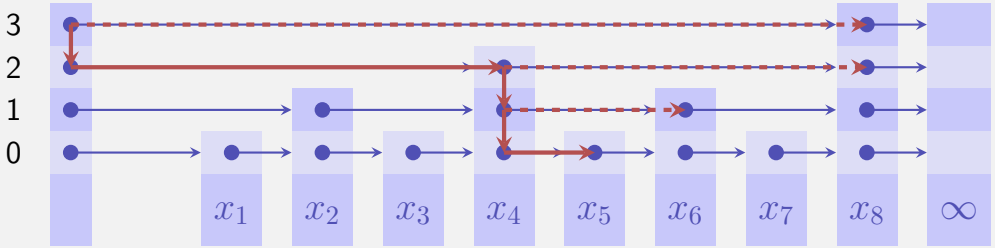
Randomisierte Skipliste: Element finden



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

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

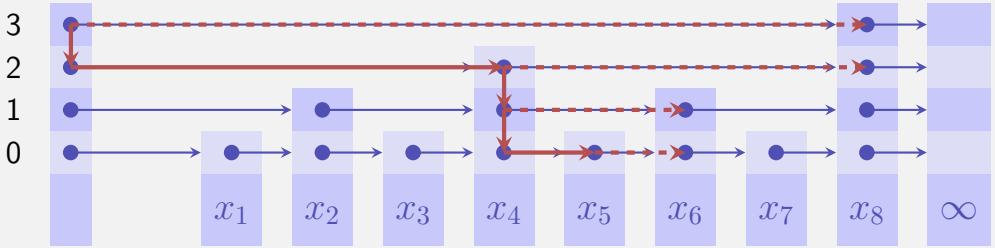
Randomisierte Skipliste: Element finden



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

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.

Implementiere `insert` and `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` and `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` and `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

being lazy “ Rule of two”:

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

Fragen?

Fragen?

Let's get to work.