

Datenstrukturen und Algorithmen

Übung 4

FS 2018

Programm von heute

- 1 Feedback letzte Übung
- 2 Wiederholung Theorie
 - Amortisierte Analyse
 - Skiplisten
- 3 Programmieraufgabe

Sortieren

Bubblesort	min	max
Vergleiche	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequenz	egal	egal
Vertauschungen	0	$\mathcal{O}(n^2)$
Sequenz	$1, 2, \dots, n$	$n, n - 1, \dots, 1$

Sortieren

InsertionSort	min	max
Vergleiche	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Sequenz	$1, 2, \dots, n$	$n, n - 1, \dots, 1$
Vertauschungen	0	$\mathcal{O}(n^2)$
Sequenz	$1, 2, \dots, n$	$n, n - 1, \dots, 1$
SelectionSort	min	max
Vergleiche	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequenz	egal	egal
Vertauschungen	0	$\mathcal{O}(n)$
Sequenz	$1, 2, \dots, n$	$n, n - 1, \dots, 1$

Sortieren

QuickSort	min	max
Vergleiche	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Sequenz	kompliziert	$1, 2, \dots, n$
Vertauschungen	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Sequenz	$1, 2, \dots, n$	kompliziert

kompliziert: Folge muss so gestaltet sein, dass der Pivot die Daten in jedem Schritt in zwei etwa gleich grosse Teile aufteilt. Zum Beispiel ($n = 7$): 4, 5, 7, 6, 2, 1, 3

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

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

A_0 : [50]

A_1 : [8, 99]

A_2 : \emptyset

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

Beispiel: einfaches Wörterbuch

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

A_0 : [50]

A_1 : [8, 99]

A_2 : \emptyset

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

Algorithmus **Find**:

Beispiel: einfaches Wörterbuch

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

A_0 : [50]

A_1 : [8, 99]

A_2 : \emptyset

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

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

Beispiel: einfaches Wörterbuch

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

A_0 : [50]

A_1 : [8, 99]

A_2 : \emptyset

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

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

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

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

Beispiel: einfaches Wörterbuch

Algorithmus `Insert(x)`:

Beispiel: einfaches Wörterbuch

Algorithmus **Insert**(x):

- Neues Array $A'_0 \leftarrow [x]$, $i \leftarrow 0$

Beispiel: einfaches Wörterbuch

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

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

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 $A_i \leftarrow A'_i$

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 $A_i \leftarrow A'_i$

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)

$A'_0 = [11]$, $A'_1 = [11, 50]$, $A'_2 = [8, 11, 50, 99]$

A_0 : [50]

A_1 : [8, 99]

A_2 : \emptyset

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

A_0 : \emptyset

A_1 : \emptyset

A_2 : [8, 11, 50, 99]

A_3 : [1, 10, 18, 20, 24, 36, 48, 75]

Insert(11)
→

Kosten Insert

Notation 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:

$$1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \dots + 2^k \cdot \frac{n}{2^k} = (k + 1)n \in \Theta(n \log n).$$

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.

⇒ 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

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

Potentialmethode

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

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

Reale Kosten:

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

Potentialmethode

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

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

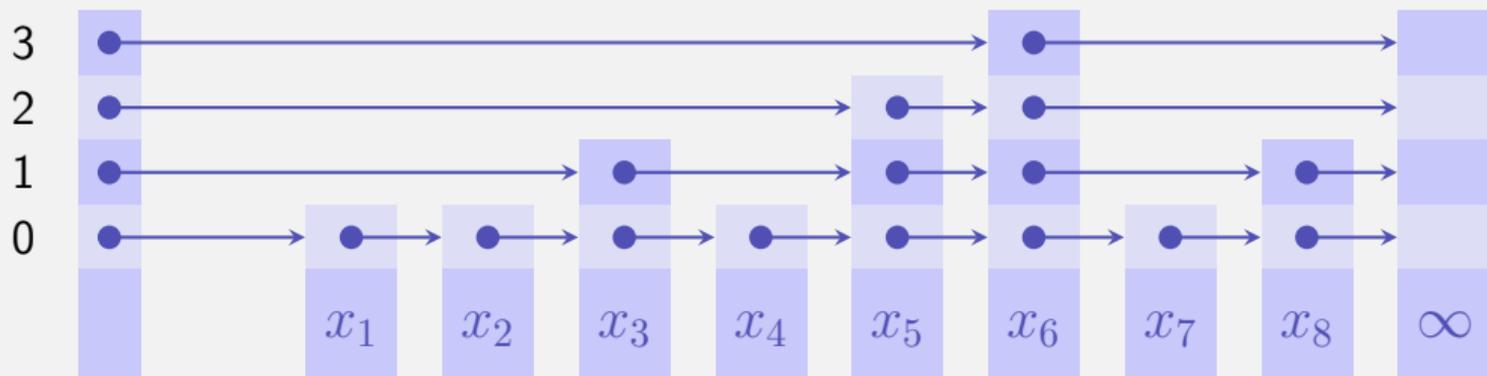
$$= (k - l) \cdot 2^l - k \cdot (2^l - 1) + l \cdot 2^l - 2^{l+1} + 2$$

$$= k - 2^{l+1} + 2$$

$$\Phi_i - \Phi_{i-1} + t_i = k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)$$

Randomisierte Skipliste

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

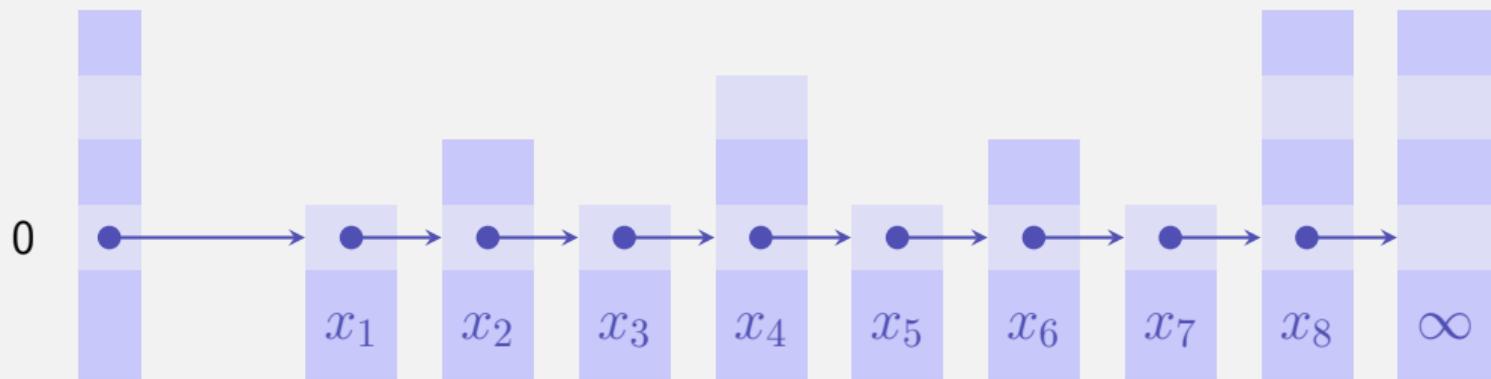


Randomisierte Skipliste: Element finden



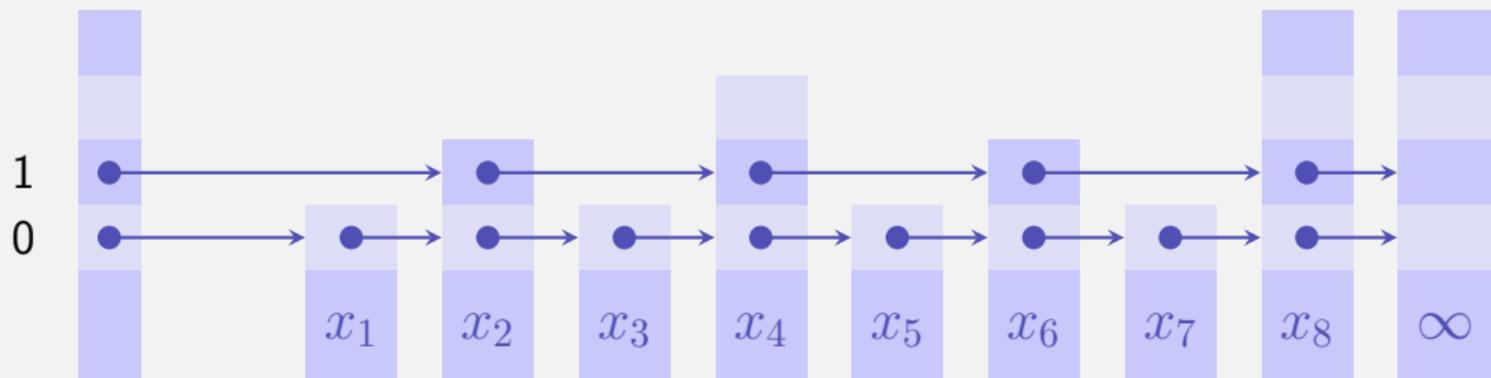
$$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9.$$

Randomisierte Skipliste: Element finden



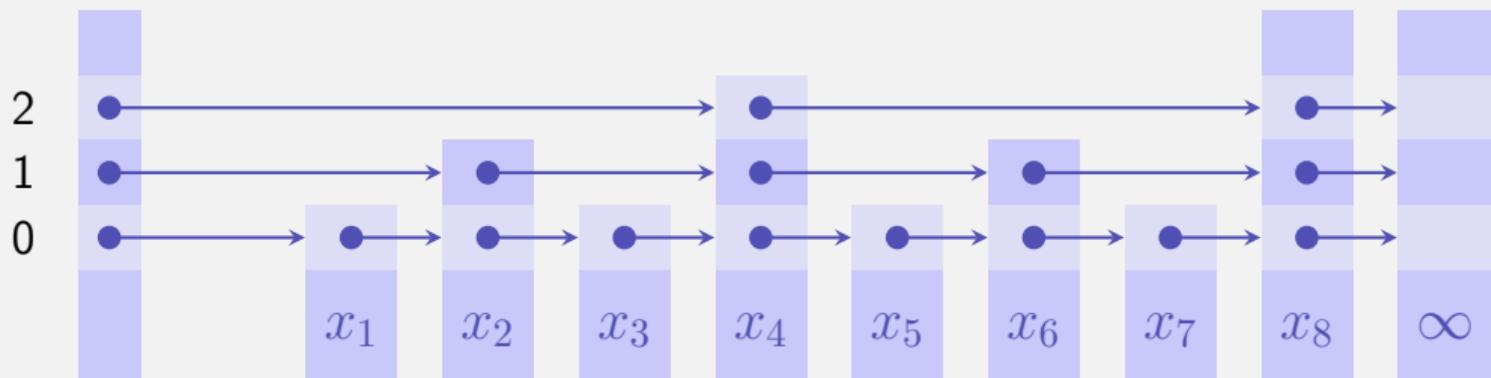
$$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9.$$

Randomisierte Skipliste: Element finden



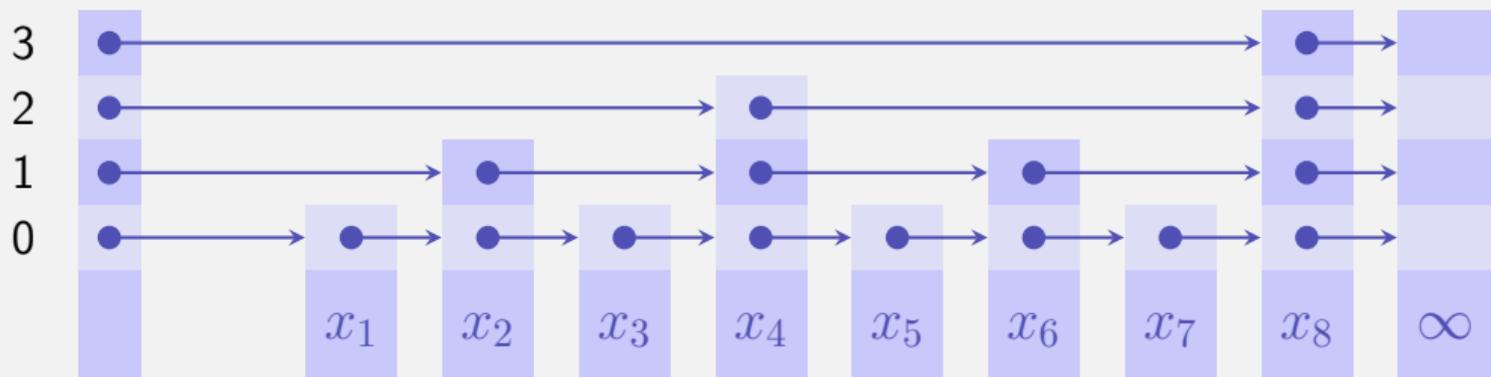
$$x_1 \leq x_2 \leq x_3 \leq \dots \leq x_9.$$

Randomisierte Skipliste: Element finden



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

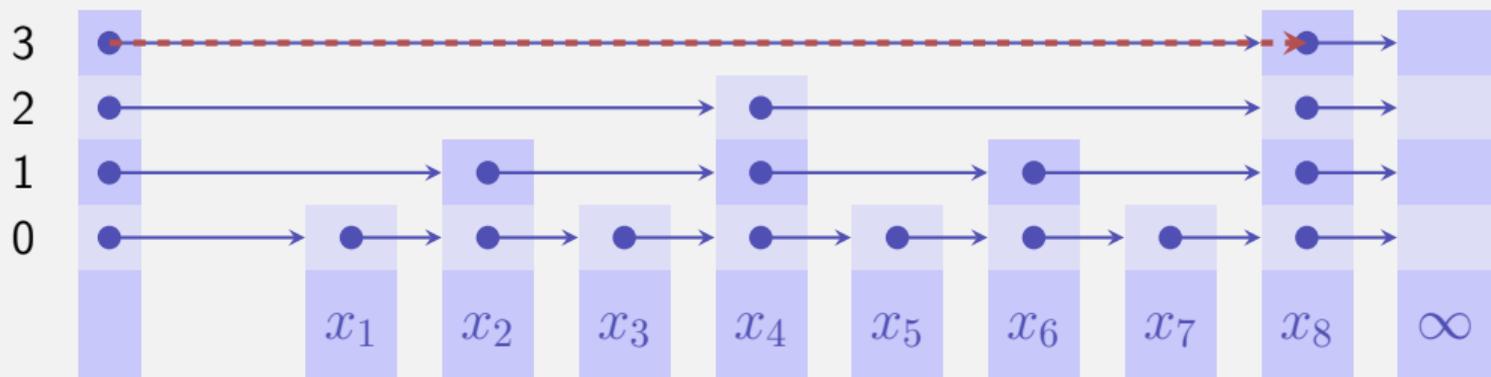
Randomisierte Skipliste: Element finden



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

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

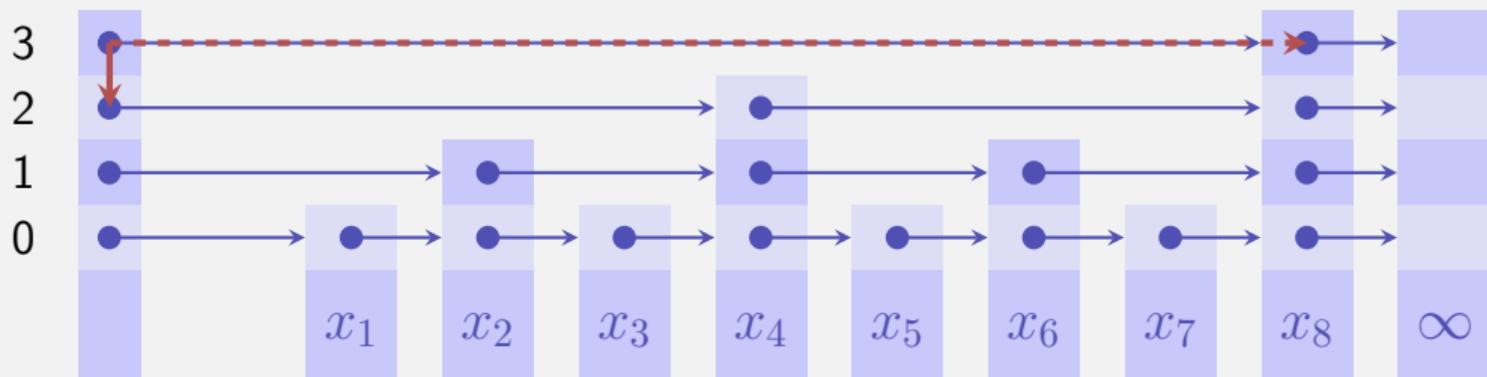
Randomisierte Skipliste: Element finden



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

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

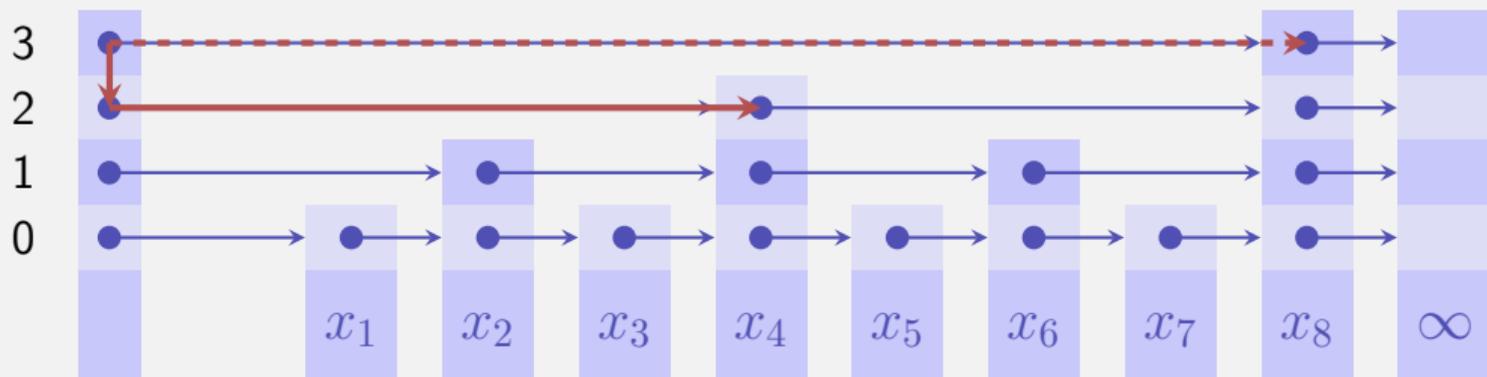
Randomisierte Skipliste: Element finden



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

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

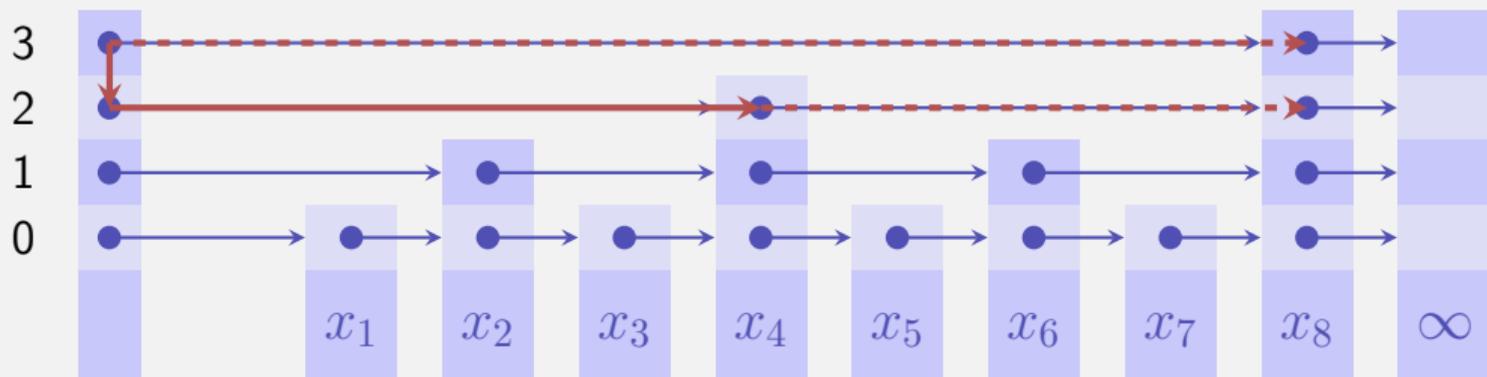
Randomisierte Skipliste: Element finden



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

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

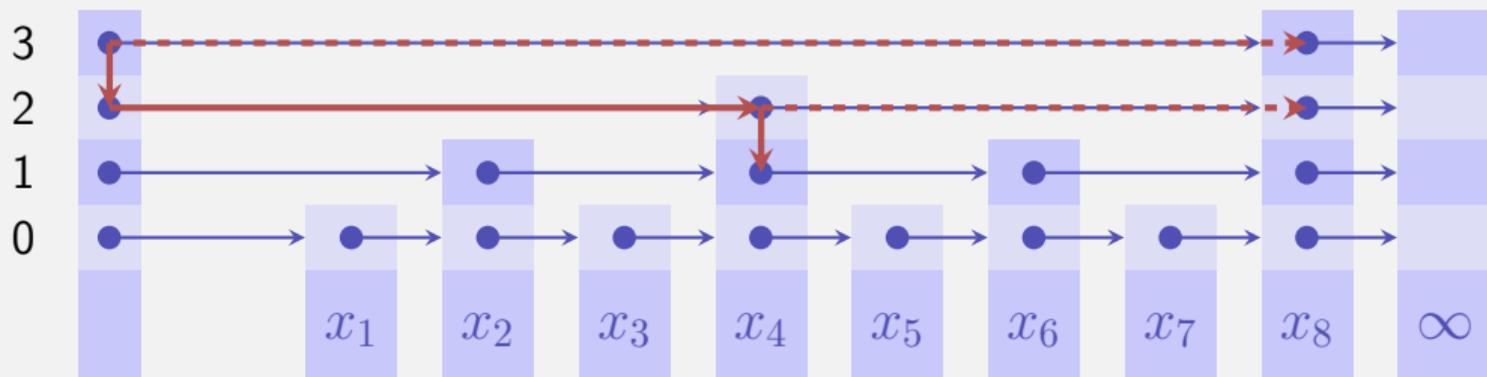
Randomisierte Skipliste: Element finden



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

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

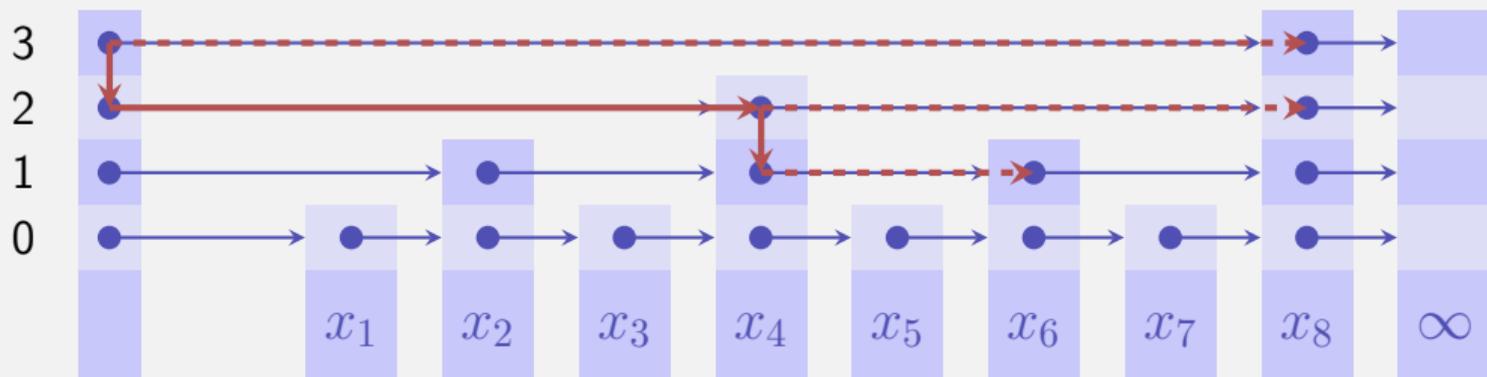
Randomisierte Skipliste: Element finden



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

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

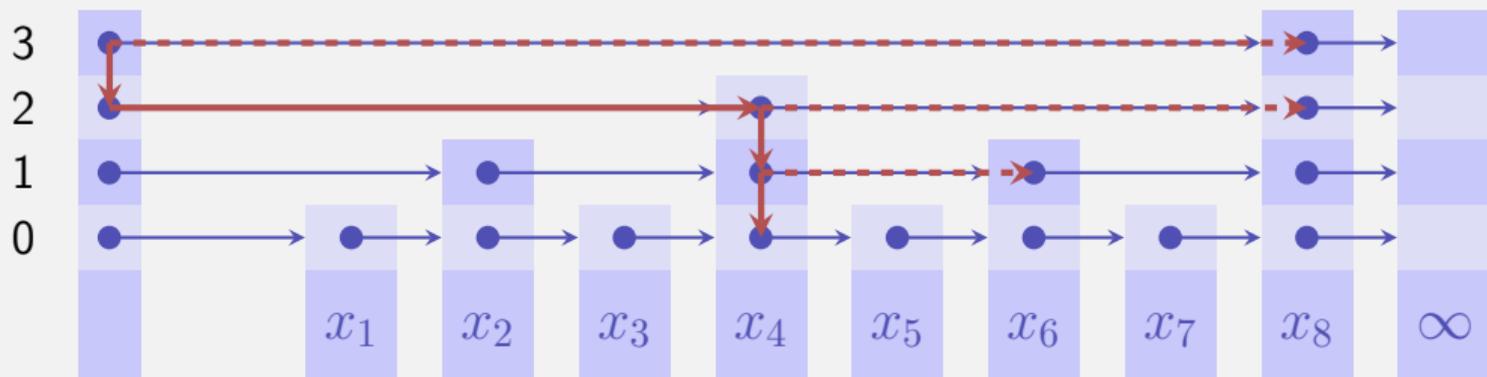
Randomisierte Skipliste: Element finden



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

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

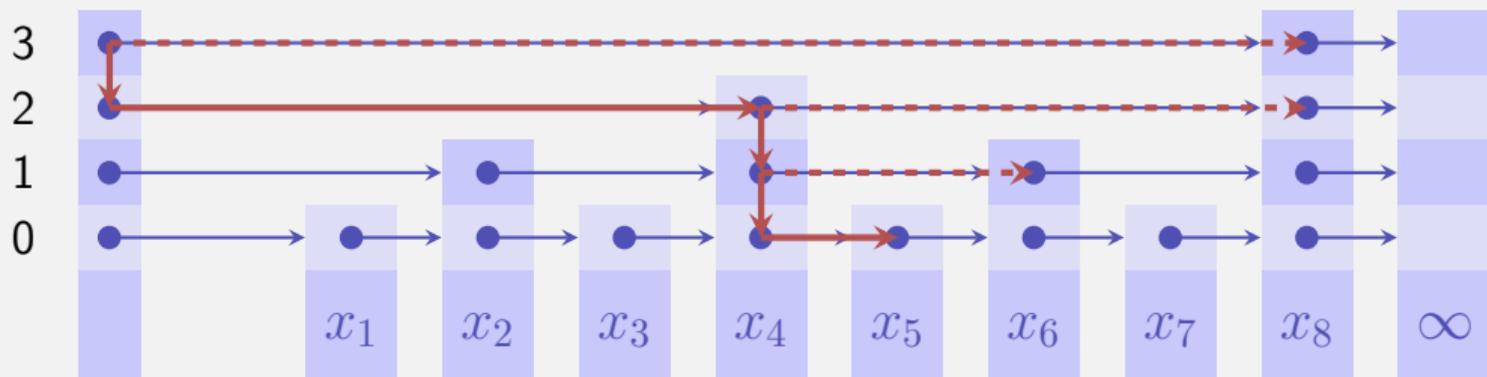
Randomisierte Skipliste: Element finden



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

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

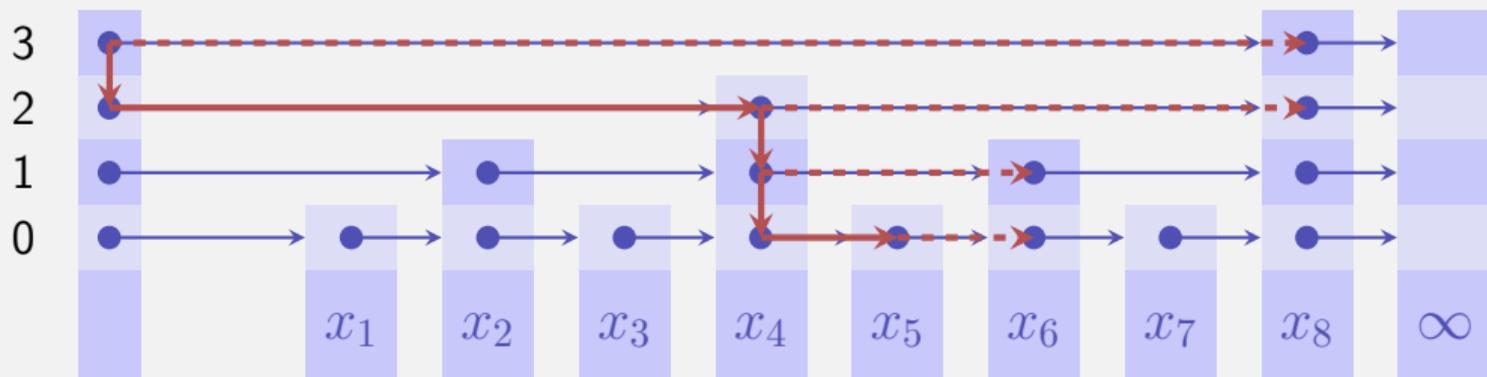
Randomisierte Skipliste: Element finden



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

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

Randomisierte Skipliste: Element finden



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

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.

Typen als Template-Parameter

```
template <typename ElementType>
class vector{
    size_t size;
    T* elem;
public:
    ...
    vector(size_t s):
    size{s},
    elem{new ElementType[s]}{}
    ...
    ElementType& operator[](size_t pos){
        return elem[pos];
    }
    ...
}
```

Funktions-Templates

```
template <typename T> // square number
T sq(T x){
    return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
    for(auto& x: c)
        x = f(x);
}
int main(){
    std::vector<int> v={1,2,3};
    apply(v,sq<int>);
    output(v); // 1 4 9
}
```

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
- gegebenenfalls 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
- gegebenenfalls Node löschen

Warnung: Es können gleiche Werte mehrfach vorkommen.

Wiederholung dynamisch allozierter Speicher

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

Wiederholung dynamisch allozierter Speicher

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

Deshalb “Rule of three”:

- constructor
- copy constructor
- destructor

Wiederholung dynamisch allozierter 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?