11. Elementare Datenstrukturen

Abstrakte Datentypen Stapel, Warteschlange, Implementationsvarianten der verketteten Liste [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

Abstrakte Datentypen

Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

[13] hoffentlich

Abstrakte Datentypen

Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

- **`push`**(x, S): Legt Element x auf den Stapel S.

[13] hoffentlich

Abstrakte Datentypen

Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

- **`push`**(x, S): Legt Element x auf den Stapel S.
- **`pop`**(S): Entfernt und liefert oberstes Element von S, oder **`null`**.

[13] hoffentlich

Abstrakte Datentypen

Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

- `push`(x, S): Legt Element x auf den Stapel S.
- `pop`(S): Entfernt und liefert oberstes Element von S, oder `null`.
- `top`(S): Liefert oberstes Element von S, oder `null`.

[13] hoffentlich

Abstrakte Datentypen

Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

- `push`(x, S): Legt Element x auf den Stapel S.
- `pop`(S): Entfernt und liefert oberstes Element von S, oder `null`.
- `top`(S): Liefert oberstes Element von S, oder `null`.
- `isEmpty`(S): Liefert `true` wenn Stack leer, sonst `false`.

[13] hoffentlich

Abstrakte Datentypen

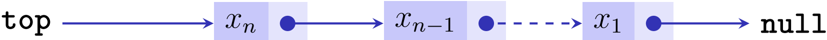
Wir erinnern uns[13] (Vorlesung Informatik I)

Ein Stack ist ein abstrakter Datentyp (ADT) mit Operationen

- **`push`**(x, S): Legt Element x auf den Stapel S.
- **`pop`**(S): Entfernt und liefert oberstes Element von S, oder **`null`**.
- **`top`**(S): Liefert oberstes Element von S, oder **`null`**.
- **`isEmpty`**(S): Liefert **`true`** wenn Stack leer, sonst **`false`**.
- **`emptyStack`**$()$: Liefert einen leeren Stack.

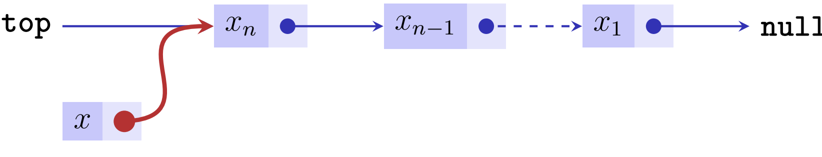
[13] hoffentlich

Implementation Push



push(x, S):

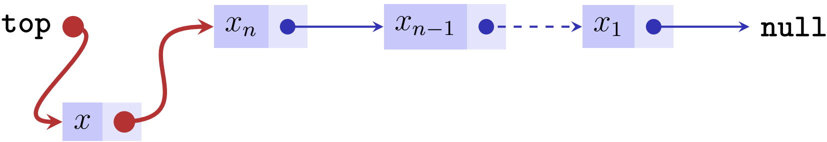
Implementation Push



push(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.

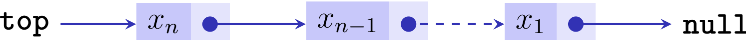
Implementation Push



push(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.
2. Setze `top` auf den Knotem mit x.

Implementation Pop



pop(S):

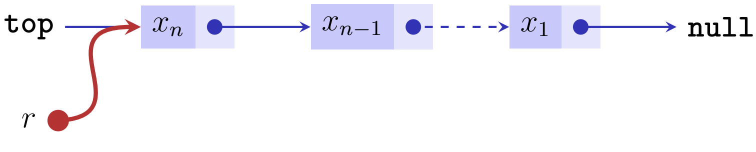
Implementation Pop

top → x_n • → x_{n-1} • - - → x_1 • → null

`pop`(S):

1. Ist `top=null`, dann gib `null` zurück

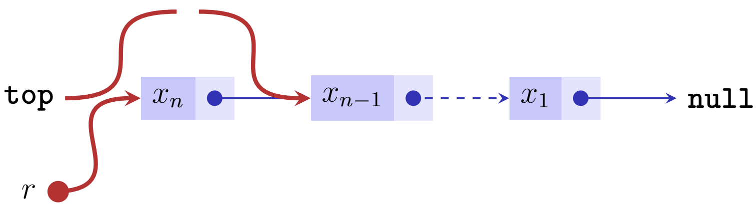
Implementation Pop



`pop`(S):

1. Ist `top=null`, dann gib `null` zurück
2. Andernfalls merke Zeiger p von `top` in r.

Implementation Pop



`pop`(S):

1. Ist `top=null`, dann gib `null` zurück
2. Andernfalls merke Zeiger p von `top` in r.
3. Setze `top` auf $p.next$ und gib r zurück

Analyse

Jede der Operationen `push`, `pop`, `top` und `isEmpty` auf dem Stack ist in $\mathcal{O}(1)$ Schritten ausführbar.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **`enqueue`**(x, Q): fügt x am Ende der Schlange an.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **`enqueue`**(x, Q): fügt x am Ende der Schlange an.
- **`dequeue`**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**`null`** sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

- **`enqueue`**(x, Q): fügt x am Ende der Schlange an.
- **`dequeue`**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**`null`** sonst.)
- **`head`**(Q): liefert das Objekt am Beginn der Schlage zurück (**`null`** sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

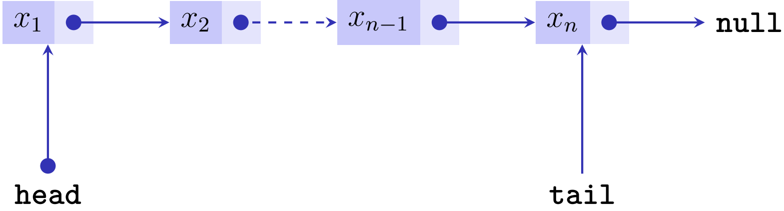
- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlage zurück (**null** sonst.)
- **isEmpty**(Q): liefert **true** wenn Queue leer, sonst **false**.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

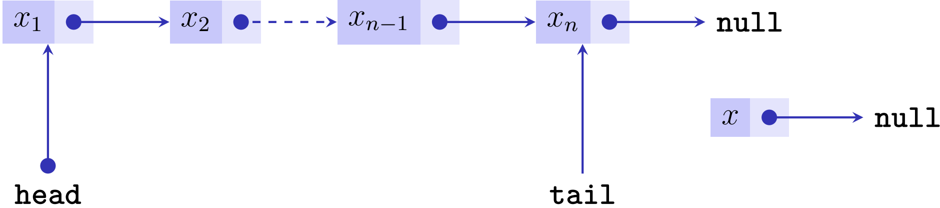
- **`enqueue`**(x, Q): fügt x am Ende der Schlange an.
- **`dequeue`**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (`null` sonst.)
- **`head`**(Q): liefert das Objekt am Beginn der Schlage zurück (`null` sonst.)
- **`isEmpty`**(Q): liefert `true` wenn Queue leer, sonst `false`.
- **`emptyQueue`**$()$: liefert leere Queue zurück.

Implementation Queue



enqueue(x, S):

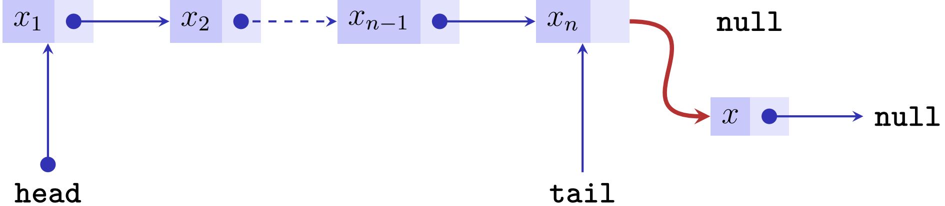
Implementation Queue



enqueue(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf **null**.

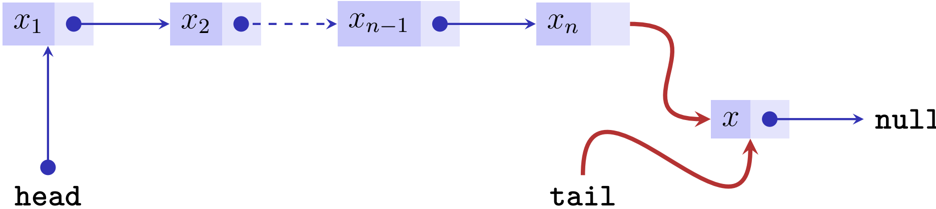
Implementation Queue



enqueue(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.

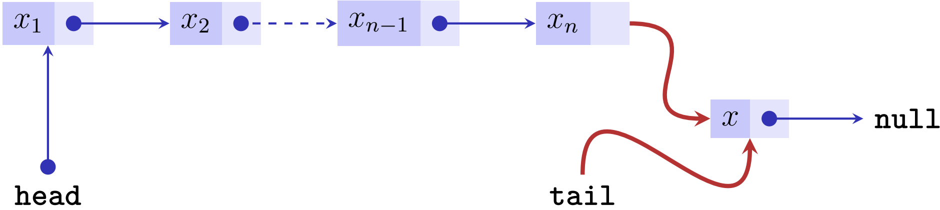
Implementation Queue



enqueue(x, S):

1. Erzeuge neues Listenelement mit x und Zeiger auf `null`.
2. Wenn `tail` $\neq$ `null` , setze `tail.next` auf den Knoten mit x.
3. Setze `tail` auf den Knoten mit x.

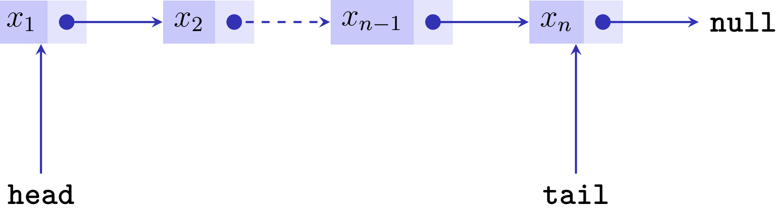
Implementation Queue



enqueue(x, S):

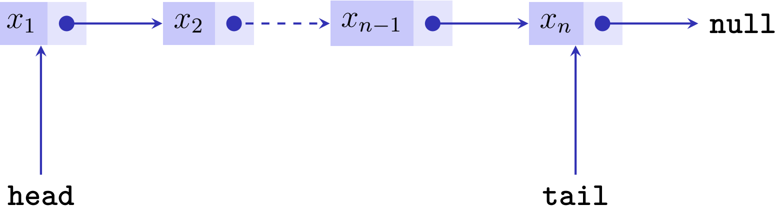
1. Erzeuge neues Listenelement mit x und Zeiger auf **null**.
2. Wenn **tail** $\neq$ **null** , setze **tail.next** auf den Knoten mit x.
3. Setze **tail** auf den Knoten mit x.
4. Ist **head** $=$ **null**, dann setze **head** auf **tail**.

Invarianten!



Mit dieser Implementation gilt

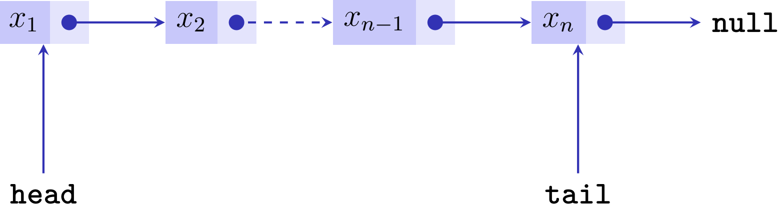
Invarianten!



Mit dieser Implementation gilt

- entweder head $=$ tail $=$ null,

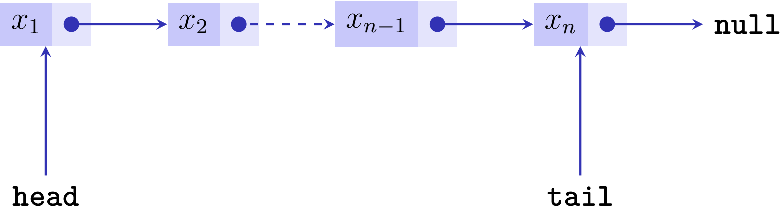
Invarianten!



Mit dieser Implementation gilt

- entweder $\texttt{head} = \texttt{tail} = \texttt{null}$,
- oder $\texttt{head} = \texttt{tail} \neq \texttt{null}$ und $\texttt{head.next} = \texttt{null}$

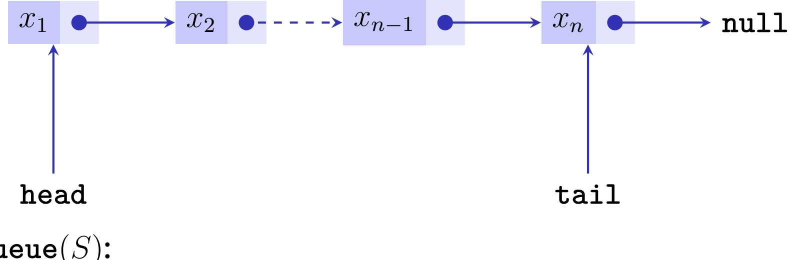
Invarianten!



Mit dieser Implementation gilt

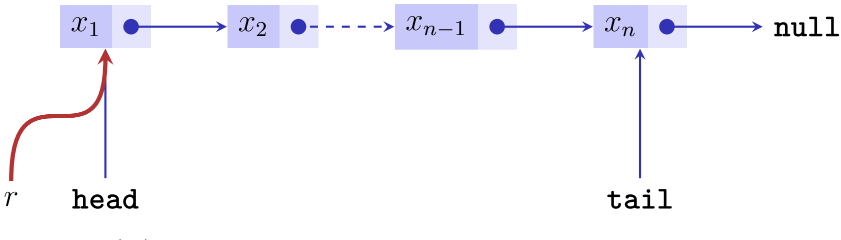
- entweder head = tail = null,
- oder head = tail ≠ null und head.next = null
- oder head ≠ null und tail ≠ null und head ≠ tail und head.next ≠ null.

Implementation Queue



dequeue(S):

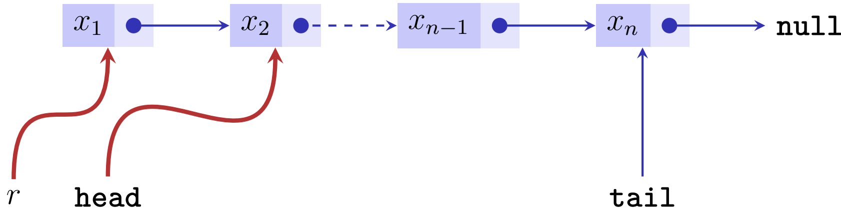
Implementation Queue



dequeue(S):

1. Merke Zeiger von **head** in r. Wenn $r =$ **null**, gib r zurück.

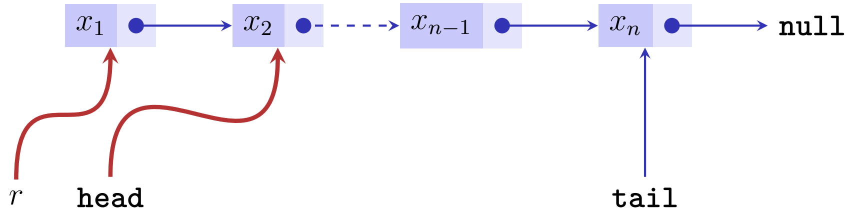
Implementation Queue



dequeue(S):

1. Merke Zeiger von **head** in r. Wenn $r =$ **null**, gib r zurück.
2. Setze den Zeiger von **head** auf **head.next**.

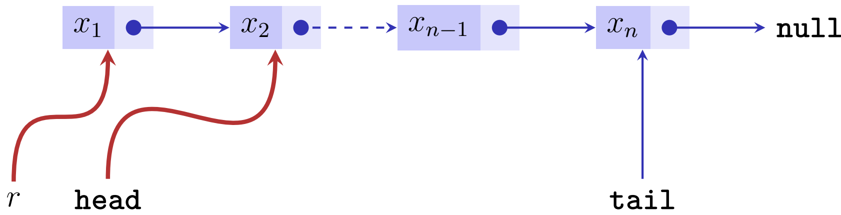
Implementation Queue



dequeue(S):

1. Merke Zeiger von `head` in r. Wenn $r =$ `null`, gib r zurück.
2. Setze den Zeiger von `head` auf `head.next`.
3. Ist nun `head` $=$ `null`, dann setze tail auf `null`.

Implementation Queue



dequeue(S):

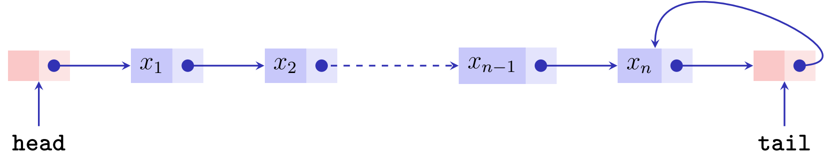
1. Merke Zeiger von **head** in r. Wenn $r =$ **null**, gib r zurück.
2. Setze den Zeiger von **head** auf **head.next**.
3. Ist nun **head** $=$ **null**, dann setze tail auf **null**.
4. Gib den Wert von r zurück.

Analyse

Jede der Operationen `enqueue`, `dequeue`, `head` und `isEmpty` auf der Queue ist in $\mathcal{O}(1)$ Schritten ausführbar.

Implementationsvarianten verketteter Listen

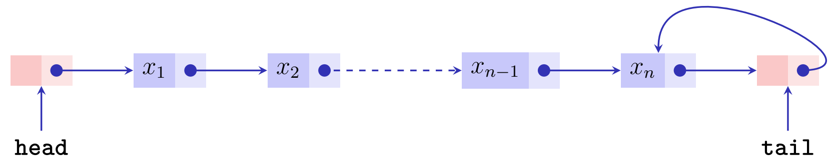
Liste mit Dummy-Elementen (Sentinels).



Vorteil: Weniger Spezialfälle!

Implementationsvarianten verketteter Listen

Liste mit Dummy-Elementen (Sentinels).

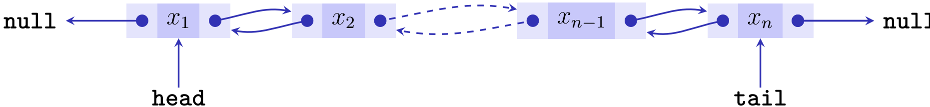


Vorteil: Weniger Spezialfälle!

Variante davon: genauso, dabei Zeiger auf ein Element immer einfach indirekt gespeichert. (Bsp: Zeiger auf x_3 zeigt auf x_2.)

Implementationsvarianten verketteter Listen

Doppelt verkettete Liste



Übersicht

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = Einfach verkettet
(B) = Einfach verkettet, mit Dummyelement am Anfang und Ende
(C) = Einfach verkettet, mit einfach indirekter Elementaddressierung
(D) = Doppelt verkettet

12. Amortisierte Analyse

Amortisierte Analyse: Aggregat Analyse, Konto-Methode, Potentialmethode [Ottman/Widmayer, Kap. 3.3, Cormen et al, Kap. 17]

Multistack

Multistack unterstützt neben `push` und `pop` noch

`multipop`(k, S): Entferne die $\min(\text{size}(S), k)$ zuletzt eingefügten Objekte und liefere diese zurück.

Implementation wie beim Stack. Laufzeit von `multipop` ist $\mathcal{O}(k)$.

Akademische Frage

Führen wir auf einem Stack mit n Elementen n mal `multipop(k,S)` aus, kostet das dann $\mathcal{O}(n^2)$?

Akademische Frage

Führen wir auf einem Stack mit n Elementen n mal `multipop(k,S)` aus, kostet das dann $\mathcal{O}(n^2)$?

Sicher richtig, denn jeder `multipop` kann Zeit $\mathcal{O}(n)$ haben.

Akademische Frage

Führen wir auf einem Stack mit n Elementen n mal `multipop(k,S)` aus, kostet das dann $\mathcal{O}(n^2)$?

Sicher richtig, denn jeder `multipop` kann Zeit $\mathcal{O}(n)$ haben.

Wie bekommen wir eine schärfere Abschätzung?

Amortisierte Analyse

- Obere Schranke: Abschätzung der durchschnittlichen Laufzeit jeder betrachteten Operation im schlechtesten Fall.

$$\frac{1}{n}\sum_{i=1}^{n} \text{Kosten}(\text{op}_i)$$

- Nutzt aus, dass wenige teure Operationen vielen billigen Operationen gegenüberstehen.
- In der amortisierten Analyse sucht man nach einer Kostenfunktion / einem Potential, um zu zeigen, wie die billigen Operationen für die teuren Operationen “aufkommen” können.

Aggregierte Analyse

Direkte Argumentation: berechne eine Schranke für die Gesamtzahl der Elementaroperationen und teile durch die Anzahl der Operationen

Aggregierte Analyse: (Stack)

- Bei n Operationen können insgesamt maximal n Elemente auf den Stack gelegt werden. Also können auch insgesamt nur maximal n Elemente vom Stack entfernt werden.
- Für die Gesamtkosten ergibt sich

$$\sum_{i=1}^{n} \text{Kosten}(\text{op}_i) \leq 2n$$

und somit

$$\text{amortisierte Kosten}(\text{op}_i) \leq 2 \in \mathcal{O}(1)$$

Kontomethode

Modell

- Der Computer basiert auf Münzen: jede Elementaroperation der Maschine kostet eine Münze.
- Für jede Operation op_k einer Datenstruktur wird eine bestimme Anzahl Münzen a_k auf eine Konto A eingezahlt: $A_k = A_{k-1} + a_k$
- Die Münzen vom Konto A werden verwendet, um die anfallenden echten Kosten t_k zu bezahlen.
- Das Konto A muss zu jeder Zeit genügend Münzen aufweisen, um die laufende Operation op_k zu bezahlen: $A_k - t_k \geq 0 \forall k$.

$\Rightarrow$ a_k sind die amortisierten Kosten der Operation op_k.

Kontomethode (Stack)

- Aufruf von `push`: kostet 1 CHF und zusätzlich kommt 1 CHF auf das Bankkonto ($a_k = 2$)
- Aufruf von `pop`: kostet 1 CHF, wird durch Rückzahlung vom Bankkonto beglichen. ($a_k = 0$)

Kontostand wird niemals negativ.

$a_k \leq 2 \,\forall\, k$, also: konstante amortisierte Kosten.

Potentialmethode

Leicht anderes Modell

- Definiere ein Potential Φ_i, welches zum Zustand der betrachteten Datenstruktur zum Zeitpunkt i gehört.
- Das Potential soll zum Ausgleichen teurer Operationen verwendet werden und muss daher so gewählt sein, dass es bei (häufigen) günstigen Operationen erhöht wird, während es die (seltenen) teuren Operationen durch einen fallenden Wert bezahlt.

Potentialmethode (formal)

Bezeichne t_i die realen Kosten der Operation op_i.
Potentialfunktion $\Phi_i \geq 0$ zur Datenstruktur nach i Operationen.
Voraussetzung: $\Phi_i \geq \Phi_0 \; \forall i$.
Amortisierte Kosten der i-ten Operation:

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

Es gilt nämlich

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

Beispiel Stack

Potentialfunktion Φ_i = Anzahl Elemente auf dem Stack.

- $\texttt{push}(x, S)$: Reale Kosten $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortisierte Kosten $a_i = 2$.
- $\texttt{pop}(S)$: Reale Kosten $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortisierte Kosten $a_i = 0$.
- $\texttt{multipop}(k, S)$: Reale Kosten $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. Amortisierte Kosten $a_i = 0$.

Alle Operationen haben konstante amortisierte Kosten! Im Durchschnitt hat also Multipop konstanten Zeitbedarf. [14]

[14] Achtung: es geht nicht um den probabilistischen Mittelwert sondern den (worst-case) Durchschnitt der Kosten.

Beispiel Binärer Zähler

Binärer Zähler mit k bits. Im schlimmsten Fall für jede Zähloperation maximal k Bitflips. Also $\mathcal{O}(n \cdot k)$ Bitflips für Zählen von 1 bis n. Geht das besser?

Reale Kosten t_i = Anzahl Bitwechsel von 0 nach 1 plus Anzahl Bitwechsel von 1 nach 0.

$$...0\underbrace{1111111}_{l\text{ Einsen}}+1 = ...1\underbrace{0000000}_{l\text{ Nullen}}.$$

$$\Rightarrow t_i = l + 1$$

Binärer Zähler: Aggregatmethode

Zähle die Anzahl Bitwechsel beim Zählen von 0 bis $n-1$.

Beobachtung

- Bit 0 wechselt für jedes $k-1 \to k$
- Bit 1 wechselt für jedes $2k-1 \to 2k$
- Bit 2 wechselt für jedes $4k-1 \to 4k$

Gesamte Anzahl Bitwechsel $\sum_{i=0}^{n-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

Amortisierte Kosten für jede Erhöhung: $\mathcal{O}(1)$ Bitwechsel.

Binärer Zähler: Kontomethode

Beobachtung: bei jedem Inkrementieren wird genau ein Bit auf 1 gesetzt, während viele Bits auf 0 gesetzt werden könnten. Nur ein vorgängig auf 1 gesetztes Bit kann wieder auf 0 zurückgesetzt werden.

$a_i = 2$: 1 CHF reale Kosten für das Setzen $0 \rightarrow 1$ plus 1 CHF für das Konto. Jedes Zurücksetzen $1 \rightarrow 0$ kann vom Konto beglichen werden.

Binärer Zähler: Potentialmethode

$$\ldots 0 \underbrace{1111111}_{l \text{ Einsen}} + 1 = \ldots 1 \underbrace{0000000}_{l \text{ Nullen}}$$

Potentialfunktion Φ_i: Anzahl der 1-Bits von x_i.

$$\Rightarrow \Phi_0 = 0 \leq \Phi_i \; \forall i$$

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortisiert konstante Kosten für eine Zähloperation. ☺

13. Wörterbücher

Wörterbuch, Selbstandornung, Implementation Wörterbuch mit Array / Liste / Skipliste. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

Wörterbuch (Dictionary)

ADT zur Verwaltung von Schlüsseln aus $\mathcal{K}$ mit Operationen

- `insert`(k, D): Hinzufügen von $k \in \mathcal{K}$ in Wörterbuch D. Bereits vorhanden $\Rightarrow$ Fehlermeldung.
- `delete`(k, D): Löschen von k aus dem Wörterbuch D. Nicht vorhanden $\Rightarrow$ Fehlermeldung.
- `search`(k, D): Liefert `true` wenn $k \in D$, sonst `false`.

Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen

Einfügen

Löschen

Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(\log n)$ 🙂
Einfügen	
Löschen	

Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(\log n)$ 🙂
Einfügen	$\mathcal{O}(n)$ 🙁
Löschen	

Idee

Implementiere Wörterbuch als sortiertes Array.

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(\log n)$ 🙂
Einfügen	$\mathcal{O}(n)$ 🙁
Löschen	$\mathcal{O}(n)$ 🙁

Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen
Einfügen
Löschen

[15] Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$ ☹
Einfügen	
Löschen	

[15] Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$ ☹
Einfügen	$\mathcal{O}(1)$[15] ☺
Löschen	

[15] Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

Andere Idee

Implementiere Wörterbuch als verkettete Liste

Anzahl Elementaroperationen im schlechtesten Fall

Suchen	$\mathcal{O}(n)$	☹
Einfügen	$\mathcal{O}(1)$[15]	☺
Löschen	$\mathcal{O}(n)$	☹

[15] Unter der Voraussetzung, dass wir die Existenz nicht prüfen wollen.

13.1 Selbstanordnung

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.

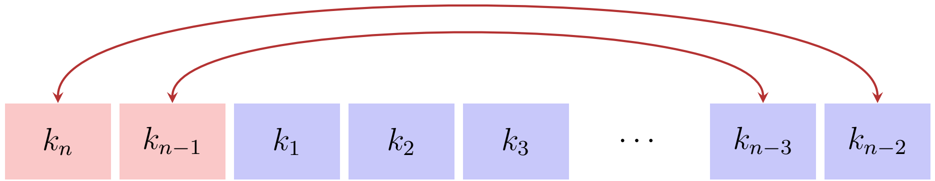
Move-to-Front

Move-to-Front:

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

Move-to-Front

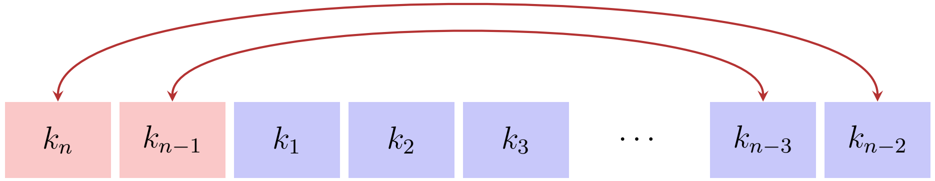
Move-to-Front:



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

Move-to-Front

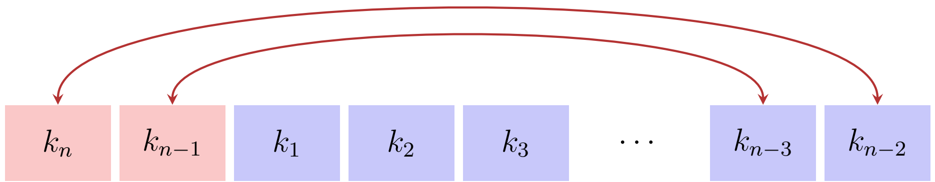
Move-to-Front:



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.

Analyse

Vergleichen MTF mit dem bestmöglichen Konkurrenten (Algorithmus) A.
Wie viel besser kann A sein?

Annahmen:

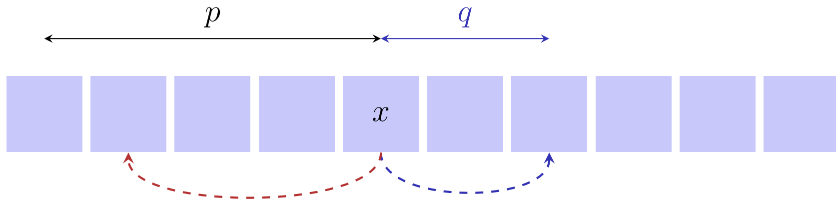
- MTF und A dürfen jeweils nur das zugegriffene Element x verschieben.
- MTF und A starten mit derselben Liste.

M_k und A_k bezeichnen die Liste nach dem k-ten Schritt. $M_0 = A_0$.

Analyse

Kosten:

- Zugriff auf x: Position p von x in der Liste.
- Keine weiteren Kosten, wenn x vor p verschoben wird.
- Weitere Kosten q für jedes Element, das x von p aus nach hinten verschoben wird.



Amortisierte Analyse

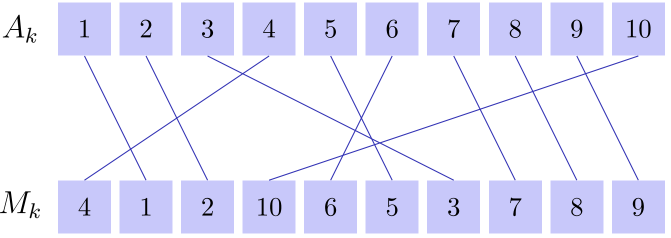
Sei eine beliebige Folge von Suchanfragen gegeben und seien $G_k^{(M)}$ und $G_k^{(A)}$ jeweils die Kosten im Schritt k für Move-to-Front und A. Suchen Abschätzung für $\sum_k G_k^{(M)}$ im Vergleich zu $\sum_k G_k^{(A)}$.

$\Rightarrow$ Amortisierte Analyse mit Potentialfunktion Φ.

Potentialfunktion

Potentialfunktion $\Phi =$ Anzahl der Inversionen von A gegen MTF.

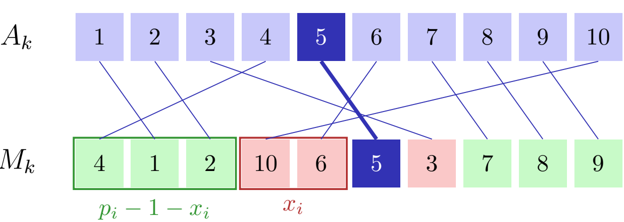
Inversion = Paar x, y so dass für die Positionen von x und y
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$

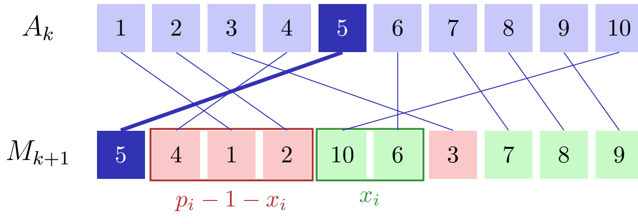


#Inversionen = #Kreuzungen

Abschätzung der Potentialfunktion: MTF

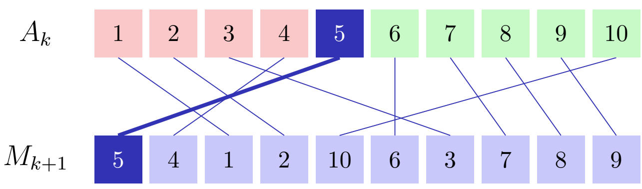
- Element i an Position $p_i := p^{(M)}(i)$.
- Zugriffskosten $C_k^{(M)} = p_i$.
- x_i: Anzahl Elemente, die in M vor p_i und in A nach i stehen.
- MTF löst x_i Inversionen auf.
- $p_i - x_i - 1$: Anzahl Elemente, die in M vor p_i und in A vor i stehen.
- MTF erzeugt $p_i - 1 - x_i$ Inversionen.

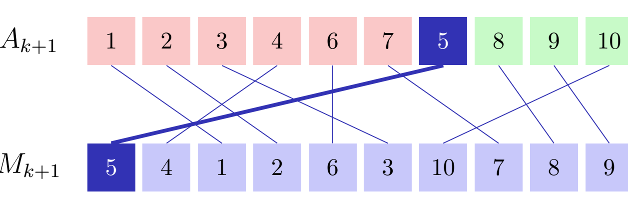




Abschätzung der Potentialfunktion: A

- Element i an Position $p^{(A)}(i)$.
- $X_k^{(A)}$: Anzahl Verschiebungen nach hinten (sonst 0).
- Zugriffskosten für i: $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$.
- A erhöht die Anzahl Inversionen höchstens um $X_k^{(A)}$.





Abschätzung

$$\Phi_{k+1} - \Phi_k \leq -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortisierte Kosten von MTF im k-ten Schritt:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\leq p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\leq C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)} \leq 2 \cdot C_k^{(A)} + X_k^{(A)}. \end{aligned}$$

Abschätzung

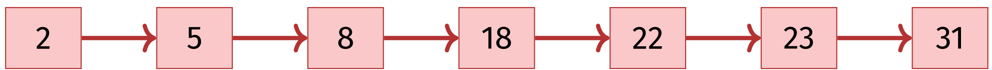
Kosten Summiert

$$\sum_k G_k^{(M)} = \sum_k C_k^{(M)} \leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)}$$
$$\leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)}$$
$$= 2 \cdot \sum_k G_k^{(A)}$$

MTF führt im schlechtesten Fall höchstens doppelt so viele Operationen aus wie eine optimale Strategie.

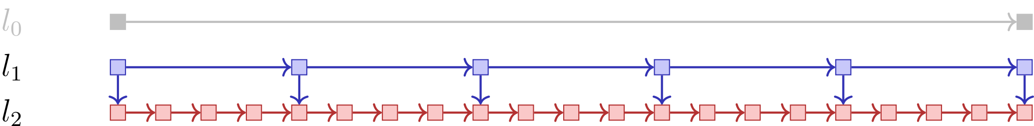
13.2 Skiplisten

Sortierte Verkettete Liste



Element / Einfügeort suchen: worst-case n Schritte.

Sortierte Verkettete Liste mit 2 Ebenen



- Anzahl Elemente: $n_0 := n$
- Schrittweite Ebene 1: n_1
- Schrittweite Ebene 2: $n_2 = 1$

$\Rightarrow$ Element / Einfügeort suchen: worst-case $\frac{n_0}{n_1} + \frac{n_1}{n_2}$.

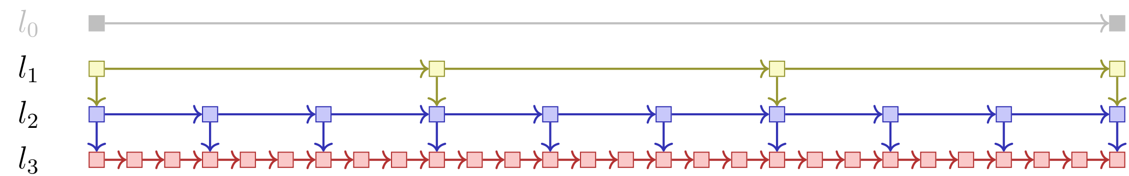
$\Rightarrow$ Beste Wahl für[16] n_1: $n_1 = \frac{n_0}{n_1} = \sqrt{n_0}$.

Element / Einfügeort suchen: worst-case $2\sqrt{n}$ Schritte.

---

[16] Differenzieren und 0 setzen, siehe Anhang

Sortierte Verkettete Liste mit 3 Ebenen



- Anzahl Elemente: $n_0 := n$
- Schrittweiten Ebenen $0 < i < 3$: n_i
- Schrittweite auf Ebene 3: $n_3 = 1$

$\Rightarrow$ Beste Wahl für (n_1, n_2): $n_2 = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \sqrt[3]{n_0}$.

Element / Einfügeort suchen: worst-case $3 \cdot \sqrt[3]{n}$ Schritte.

Sortierte Verkettete Liste mit k Ebenen (Skipliste)

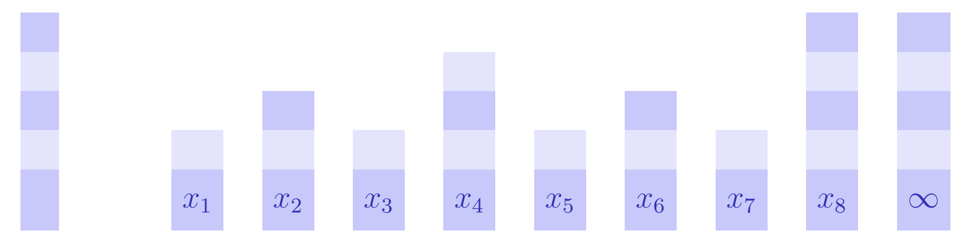
- Anzahl Elemente: $n_0 := n$
- Schrittweiten Ebenen $0 < i < k$: n_i
- Schrittweite auf Ebene k: $n_k = 1$

$\Rightarrow$ Beste Wahl für $(n_1, \ldots, n_k)$: $n_{k-1} = \frac{n_0}{n_1} = \frac{n_1}{n_2} = \cdots = \sqrt[k]{n_0}$.
Element / Einfügeort suchen: worst-case $k \cdot \sqrt[k]{n}$ Schritte[17].
Annahme: $n = 2^k$
$\Rightarrow$ worst case $\log_2 n \cdot 2$ Schritte und $\frac{n_i}{n_{i+1}} = 2 \,\forall\, 0 \leq i < \log_2 n$.

[17](Herleitung: Anhang)

Suche in Skipliste

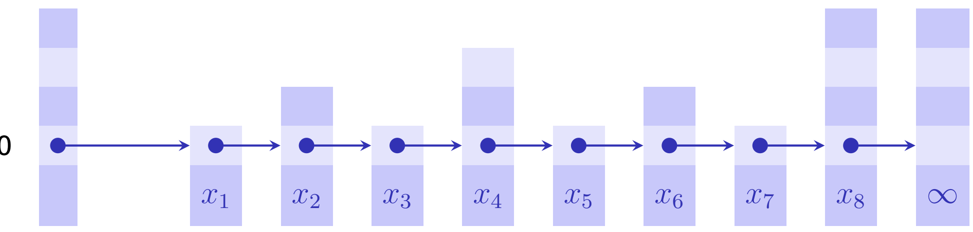
Skipliste



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

Suche in Skipliste

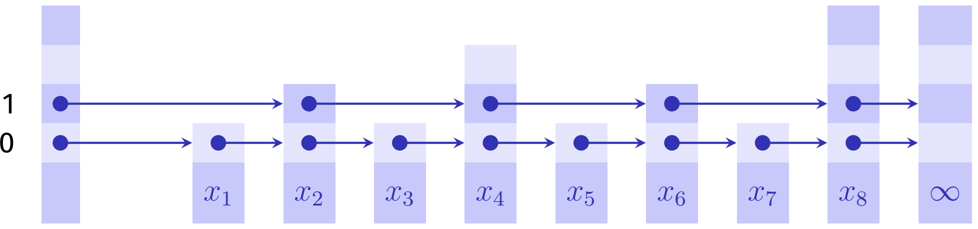
Skipliste



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

Suche in Skipliste

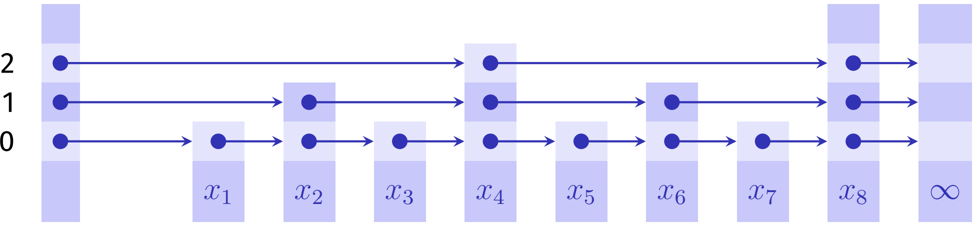
Skipliste



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

Suche in Skipliste

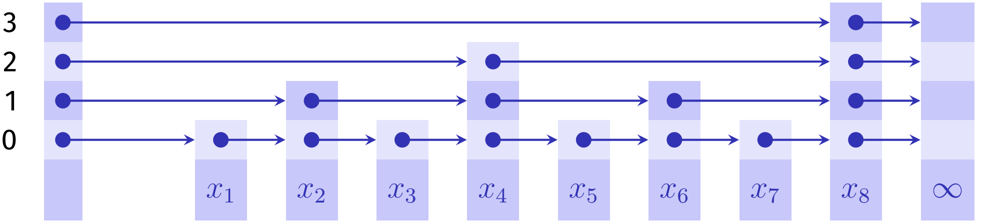
Skipliste



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

Suche in Skipliste

Perfekte Skipliste

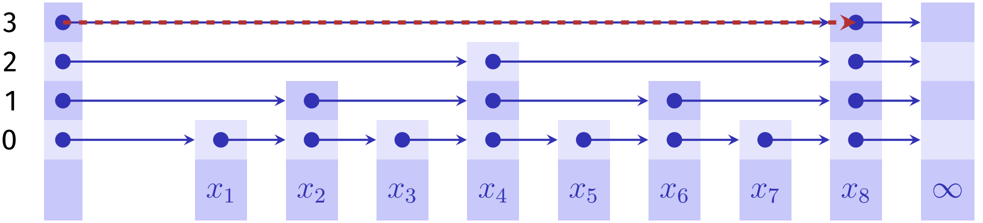


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

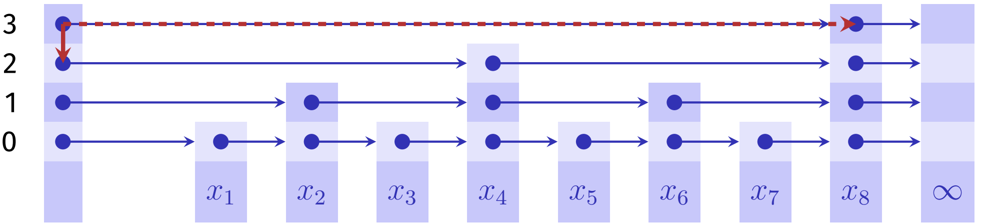


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

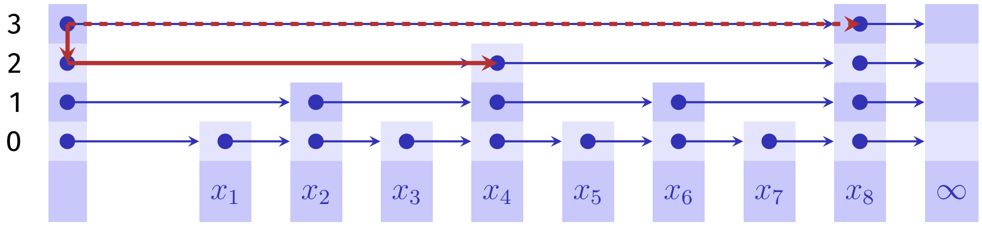


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

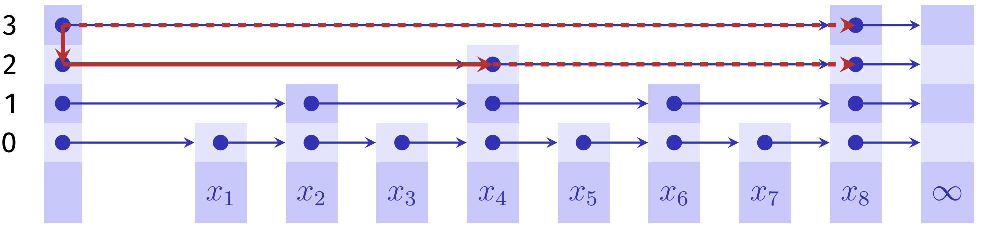


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

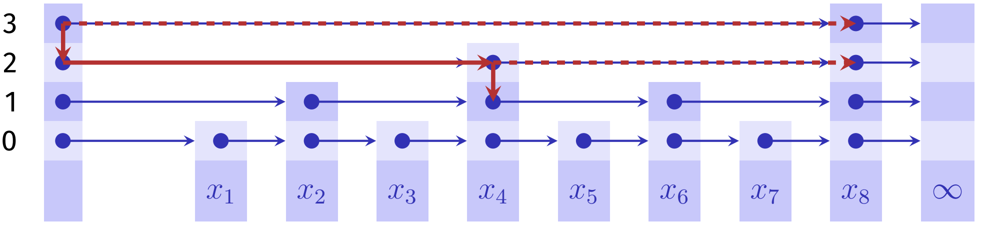


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

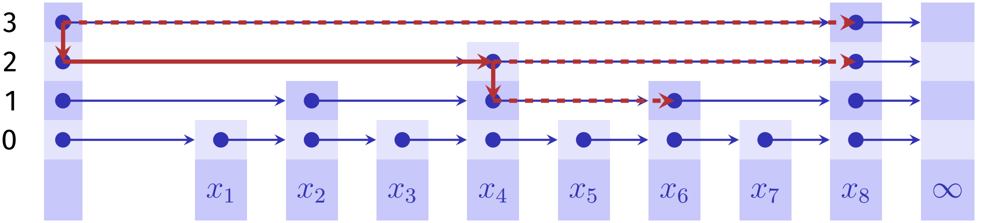


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

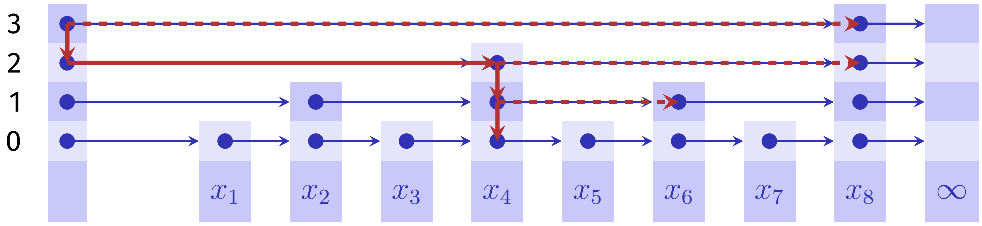


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

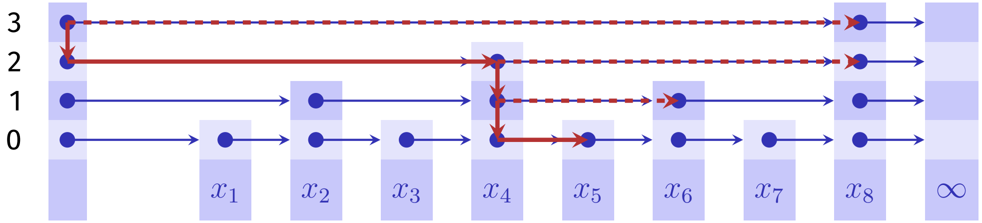


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste

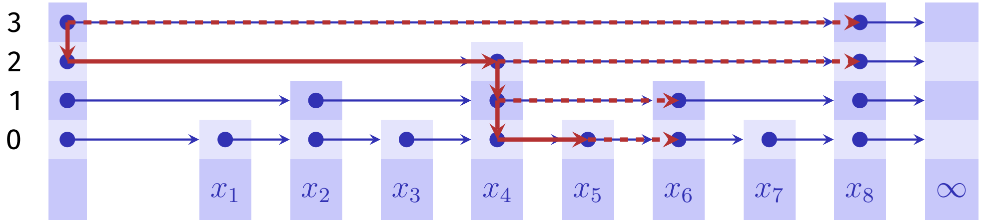


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

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

Suche in Skipliste

Perfekte Skipliste



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

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

Analyse Perfekte Skipliste (schlechtester Fall)

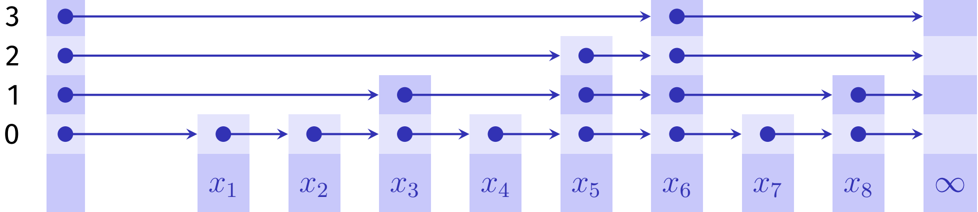
Suchen in $\mathcal{O}(\log n)$. Einfügen in $\mathcal{O}(n)$.

Randomisierte Skipliste

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

Randomisierte Skipliste

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



Analyse Randomisierte Skipliste

Theorem 15

Die Anzahl an erwarteten Elementaroperationen für Suchen, Einfügen und Löschen eines Elements in einer randomisierten Skipliste ist $\mathcal{O}(\log n)$.

Der längliche Beweis, welcher im Rahmen dieser Vorlesung nicht geführt wird, betrachtet die Länge eines Weges von einem gesuchten Knoten zurück zum Startpunkt im höchsten Level.

13.3 Anhang

Mathematik zur Skipliste

[Mathematik zur k-Level Skipliste]

Gegeben Anzahl Datenpunkte n_0, Anzahl Level $k > 0$ und Anzahl Elemente n_l die pro Level l übersprungen werden, $n_k = 1$. Maximale Anzahl totaler Schritte in der Skipliste:

$$f(\vec{n}) = \frac{n_0}{n_1} + \frac{n_1}{n_2} + \dots \frac{n_{k-1}}{n_k}$$

Minimiere f für $(n_1, \dots, n_{k-1})$: $\frac{\partial f(\vec{n})}{\partial n_t} = 0$ für alle $0 < t < k$,
$\frac{\partial f(\vec{n})}{\partial n_t} = -\frac{n_{t-1}}{{n_t}^2} + \frac{1}{n_{t+1}} = 0 \Rightarrow n_{t+1} = \frac{n_t^2}{n_{t-1}}$ und $\frac{n_{t+1}}{n_t} = \frac{n_t}{n_{t-1}}$.

[Mathematik zur k-Level Skipliste]

Vorige Folie $\Rightarrow \frac{n_t}{n_0} = \frac{n_t}{n_{t-1}} \frac{n_{t-1}}{n_{t-2}} \dots \frac{n_1}{n_0} = \left(\frac{n_1}{n_0}\right)^t$

Insbesondere $1 = n_k = \frac{n_1^k}{n_0^{k-1}} \Rightarrow n_1 = \sqrt[k]{n_0^{k-1}}$

Also $n_{k-1} = \frac{n_0}{n_1} = \sqrt[k]{\frac{n_0^k}{n_0^{k-1}}} = \sqrt[k]{n_0}$.

Maximale Anzahl Schritte in der Skipliste $f(\vec{n}) = k \cdot (\sqrt[k]{n_0})$

Angenommen $n_0 = 2^k$, dann $\frac{n_l}{n_{l+1}} = 2$ für alle $0 \leq l < k$ (Skipliste halbiert die Daten in jedem Level), und $f(n) = k \cdot 2 = 2\log_2 n \in \Theta(\log n)$.