

10. Sortieren III

Untere Schranken für das vergleichsbasierte Sortieren, Radix- und Bucketsort

10.1 Untere Grenzen für Vergleichbasiertes Sortieren

[Ottman/Widmayer, Kap. 2.8, Cormen et al, Kap. 8.1]

Untere Schranke für das Sortieren

Bis hierher: Sortieren im schlechtesten Fall benötigt $\Omega(n \log n)$ Schritte.

Geht es besser?

Untere Schranke für das Sortieren

Bis hierher: Sortieren im schlechtesten Fall benötigt $\Omega(n \log n)$ Schritte.

Geht es besser? Nein:

Theorem

Vergleichsbasierte Sortierverfahren benötigen im schlechtesten Fall und im Mittel mindestens $\Omega(n \log n)$ Schlüsselvergleiche.

Vergleichsbasiertes Sortieren

- Algorithmus muss unter $n!$ vielen Anordnungsmöglichkeiten einer Folge $(A_i)_{i=1,\dots,n}$ die richtige identifizieren.

Vergleichsbasiertes Sortieren

- Algorithmus muss unter $n!$ vielen Anordnungsmöglichkeiten einer Folge $(A_i)_{i=1,\dots,n}$ die richtige identifizieren.
- Zu Beginn weiss der Algorithmus nichts.

Vergleichsbasiertes Sortieren

- Algorithmus muss unter $n!$ vielen Anordnungsmöglichkeiten einer Folge $(A_i)_{i=1,\dots,n}$ die richtige identifizieren.
- Zu Beginn weiss der Algorithmus nichts.
- Betrachten den “Wissensgewinn” des Algorithmus als Entscheidungsbaum:

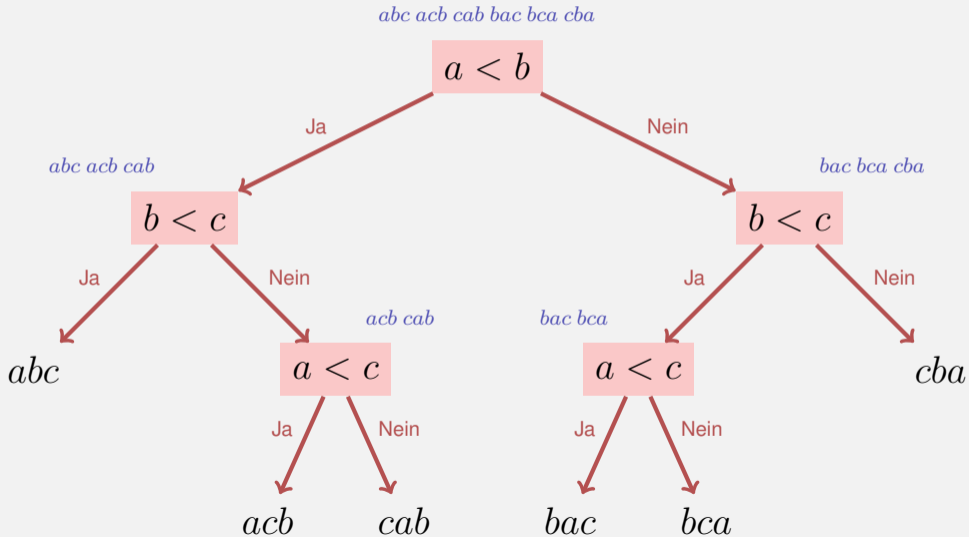
Vergleichsbasiertes Sortieren

- Algorithmus muss unter $n!$ vielen Anordnungsmöglichkeiten einer Folge $(A_i)_{i=1,\dots,n}$ die richtige identifizieren.
- Zu Beginn weiss der Algorithmus nichts.
- Betrachten den “Wissensgewinn” des Algorithmus als Entscheidungsbaum:
 - Knoten enthalten verbleibende Möglichkeiten

Vergleichsbasiertes Sortieren

- Algorithmus muss unter $n!$ vielen Anordnungsmöglichkeiten einer Folge $(A_i)_{i=1,\dots,n}$ die richtige identifizieren.
- Zu Beginn weiss der Algorithmus nichts.
- Betrachten den “Wissensgewinn” des Algorithmus als Entscheidungsbaum:
 - Knoten enthalten verbleibende Möglichkeiten
 - Kanten enthalten Entscheidungen

Entscheidungsbaum



Entscheidungsbaum

Die Höhe eines binären Baumes mit L Blättern ist mindestens $\log_2 L$. \Rightarrow Höhe des Entscheidungsbaumes $h \geq \log n! \in \Omega(n \log n)$.¹¹

Somit auch die Länge des längsten Pfades im Entscheidungsbaum $\in \Omega(n \log n)$.

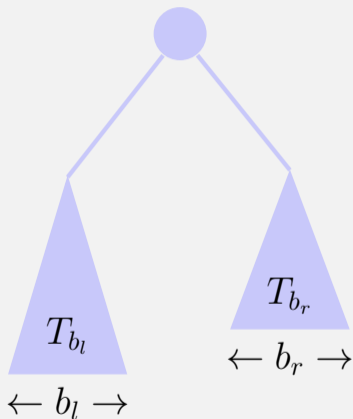
Bleibt zu zeigen: mittlere Länge $M(n)$ eines Pfades $M(n) \in \Omega(n \log n)$.

¹¹ $\log n! \in \Theta(n \log n)$:

$$\log n! = \sum_{k=1}^n \log k \leq n \log n.$$

$$\log n! = \sum_{k=1}^n \log k \geq \sum_{k=n/2}^n \log k \geq \frac{n}{2} \cdot \log \frac{n}{2}.$$

Untere Schranke im Mittel



- Entscheidungsbaum T_n mit n Blättern, mittlere Tiefe eines Blatts $m(T_n)$
- Annahme: $m(T_n) \geq \log n$ nicht für alle n .
- Wähle kleinstes b mit $m(T_b) < \log n \Rightarrow b \geq 2$
- $b_l + b_r = b$, oBdA $b_l > 0$ und $b_r > 0 \Rightarrow b_l < b, b_r < b \Rightarrow m(T_{b_l}) \geq \log b_l$ und $m(T_{b_r}) \geq \log b_r$

Untere Schranke im Mittel

Mittlere Tiefe eines Blatts:

$$\begin{aligned}m(T_b) &= \frac{b_l}{b}(m(T_{b_l}) + 1) + \frac{b_r}{b}(m(T_{b_r}) + 1) \\ &\geq \frac{1}{b}(b_l(\log b_l + 1) + b_r(\log b_r + 1)) = \frac{1}{b}(b_l \log 2b_l + b_r \log 2b_r) \\ &\geq \frac{1}{b}(b \log b) = \log b.\end{aligned}$$

Widerspruch. ■

Die letzte Ungleichung gilt, da $f(x) = x \log x$ konvex ist und für eine konvexe Funktion gilt $f((x+y)/2) \leq 1/2f(x) + 1/2f(y)$ ($x = 2b_l, y = 2b_r$ einsetzen).¹²
Einsetzen von $x = 2b_l, y = 2b_r$, und $b_l + b_r = b$.

¹²allgemein $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ für $0 \leq \lambda \leq 1$.

10.2 Radixsort und Bucketsort

Radixsort, Bucketsort [Ottman/Widmayer, Kap. 2.5, Cormen et al, Kap. 8.3]

Radix Sort

Vergleichsbasierte Sortierverfahren: Schlüssel vergleichbar ($<$ oder $>$, $=$). Ansonsten keine Voraussetzung.

Radix Sort

Vergleichsbasierte Sortierverfahren: Schlüssel vergleichbar ($<$ oder $>$, $=$). Ansonsten keine Voraussetzung.

Andere Idee: nutze mehr Information über die Zusammensetzung der Schlüssel.

Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit m Elementen.

m heisst die Wurzel (lateinisch *Radix*) der Darstellung.

Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit m Elementen.

Beispiele

m heisst die Wurzel (lateinisch *Radix*) der Darstellung.

Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit m Elementen.

Beispiele

$m = 10$ Dezimalzahlen $183 = 183_{10}$

m heisst die Wurzel (lateinisch *Radix*) der Darstellung.

Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit m Elementen.

Beispiele

$m = 10$	Dezimalzahlen	$183 = 183_{10}$
$m = 2$	Dualzahlen	101_2
$m = 16$	Hexadezimalzahlen	$A0_{16}$

m heisst die Wurzel (lateinisch *Radix*) der Darstellung.

Annahmen

Annahme: Schlüssel darstellbar als Wörter aus einem Alphabet mit m Elementen.

Beispiele

$m = 10$	Dezimalzahlen	$183 = 183_{10}$
$m = 2$	Dualzahlen	101_2
$m = 16$	Hexadezimalzahlen	$A0_{16}$
$m = 26$	Wörter	“INFORMATIK”

m heisst die Wurzel (lateinisch *Radix*) der Darstellung.

Annahmen

- Schlüssel = m -adische Zahlen mit gleicher Länge.

Annahmen

- Schlüssel = m -adische Zahlen mit gleicher Länge.
- Verfahren z zur Extraktion der k -ten Ziffer eines Schlüssels in $\mathcal{O}(1)$ Schritten.

Annahmen

- Schlüssel = m -adische Zahlen mit gleicher Länge.
- Verfahren z zur Extraktion der k -ten Ziffer eines Schlüssels in $\mathcal{O}(1)$ Schritten.

Beispiel

$$z_{10}(0, 85) = 5$$

$$z_{10}(1, 85) = 8$$

$$z_{10}(2, 85) = 0$$

Radix-Exchange-Sort

Schlüssel mit Radix 2.

Beobachtung: Wenn $k \geq 0$,

$$z_2(i, x) = z_2(i, y) \text{ für alle } i > k$$

und

$$z_2(k, x) < z_2(k, y),$$

dann $x < y$.

Radix-Exchange-Sort

Idee:

- Starte mit maximalem k .
- Binäres Aufteilen der Datensätze mit $z_2(k, \cdot) = 0$ vs. $z_2(k, \cdot) = 1$ wie bei Quicksort.
- $k \leftarrow k - 1$.

Radix-Exchange-Sort

0111 0110 1000 0011 0001

Radix-Exchange-Sort

0111 0110 1000 0011 0001

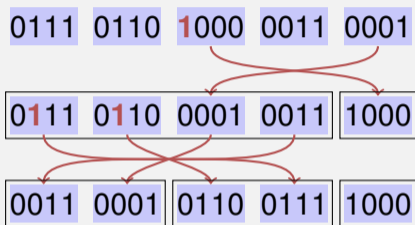
Radix-Exchange-Sort



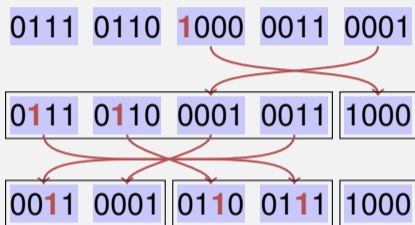
Radix-Exchange-Sort



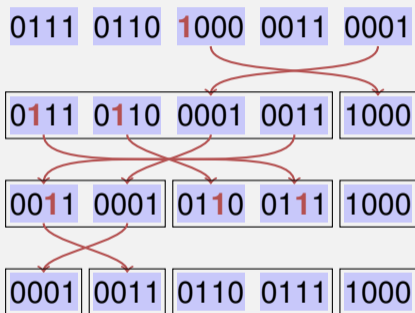
Radix-Exchange-Sort



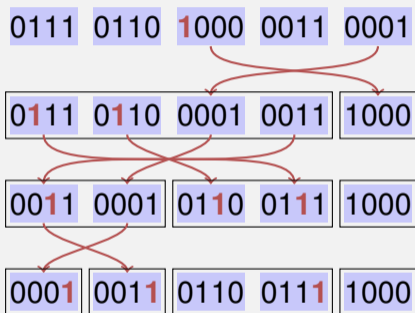
Radix-Exchange-Sort



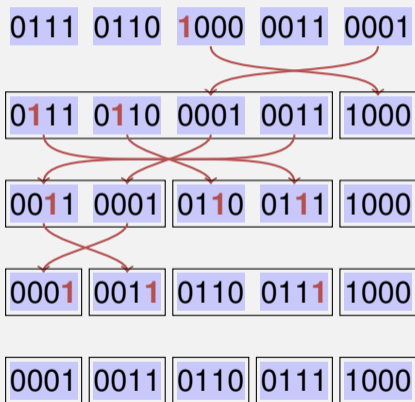
Radix-Exchange-Sort



Radix-Exchange-Sort



Radix-Exchange-Sort



Algorithmus RadixExchangeSort(A, l, r, b)

Input : Array A der Länge n , linke und rechte Grenze $1 \leq l \leq r \leq n$,
Bitposition b

Output : Array A , im Bereich $[l, r]$ nach Bits $[0, \dots, b]$ sortiert.

if $l > r$ **and** $b \geq 0$ **then**

$i \leftarrow l - 1$

$j \leftarrow r + 1$

repeat

repeat $i \leftarrow i + 1$ **until** $z_2(b, A[i]) = 1$ **and** $i \geq j$

repeat $j \leftarrow j + 1$ **until** $z_2(b, A[j]) = 0$ **and** $i \geq j$

if $i < j$ **then** swap($A[i], A[j]$)

until $i \geq j$

 RadixExchangeSort($A, l, i - 1, b - 1$)

 RadixExchangeSort($A, i, r, b - 1$)

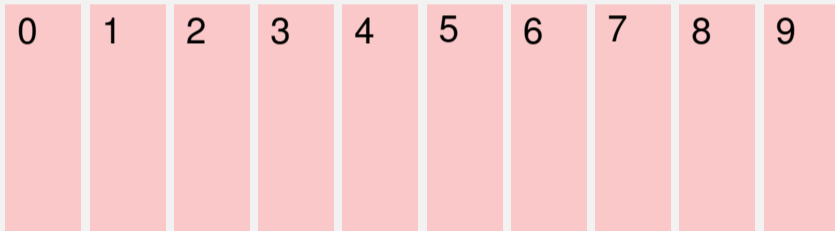
Analyse

RadixExchangeSort ist rekursiv mit maximaler Rekursionstiefe = maximaler Anzahl Ziffern p .

Laufzeit im schlechtesten Fall $\mathcal{O}(p \cdot n)$.

Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



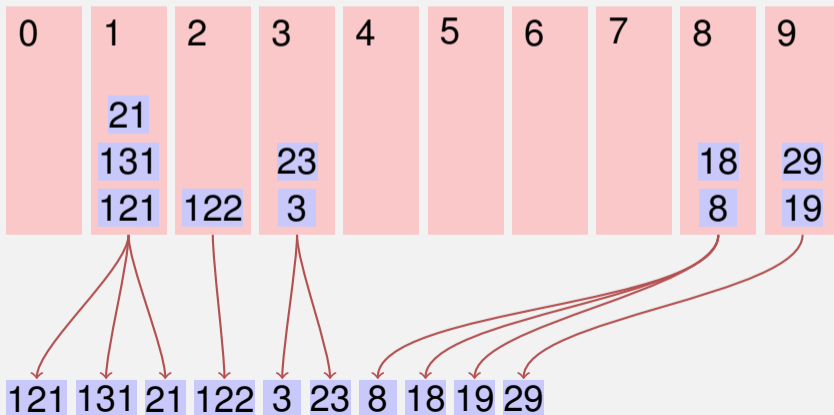
Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29



Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 122 121 131 23 21 19 29

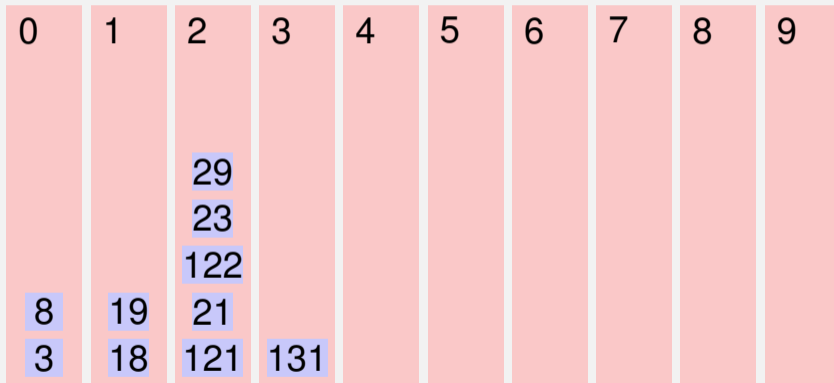


Bucket Sort (Sortieren durch Fachverteilen)

121 131 21 122 3 23 8 18 19 29

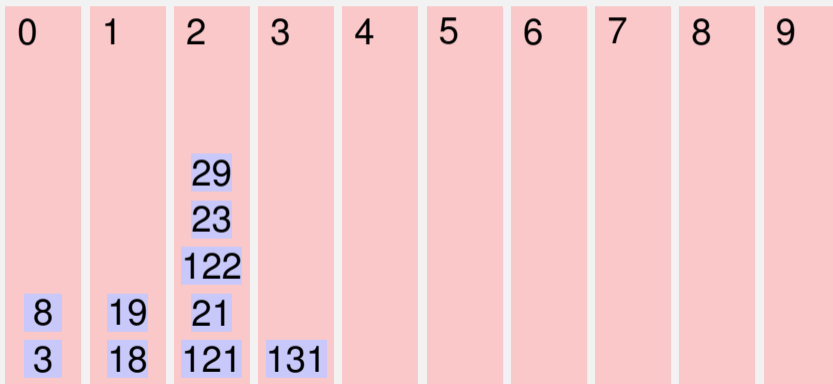
Bucket Sort (Sortieren durch Fachverteilen)

121 131 21 122 3 23 8 18 19 29



Bucket Sort (Sortieren durch Fachverteilen)

121 131 21 122 3 23 8 18 19 29



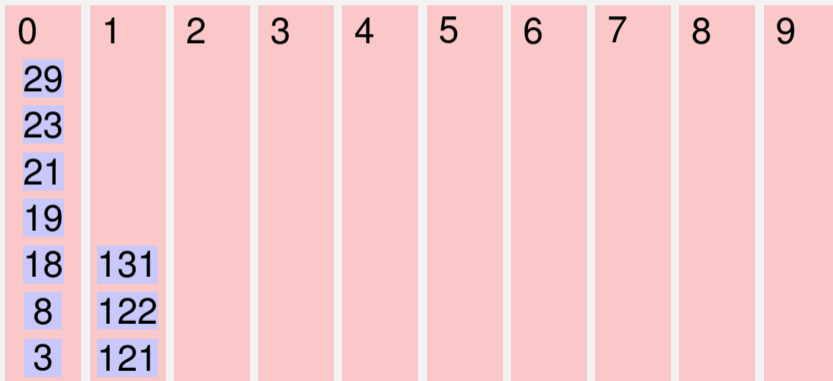
3 8 18 19 121 21 122 23 29

Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 19 121 21 122 23 29

Bucket Sort (Sortieren durch Fachverteilen)

3 8 18 19 121 21 122 23 29



3 8 18 19 21 23 29 121 122 131 😊

Implementationsdetails

Bucketgrösse sehr unterschiedlich. Zwei Möglichkeiten

Implementationsdetails

Bucketgrösse sehr unterschiedlich. Zwei Möglichkeiten

- Verkettete Liste für jede Ziffer.

Implementationsdetails

Bucketgrösse sehr unterschiedlich. Zwei Möglichkeiten

- Verkettete Liste für jede Ziffer.
- Ein Array der Länge n , Offsets für jede Ziffer in erstem Durchlauf bestimmen.

11. Elementare Datentypen

Abstrakte Datentypen Stapel, Warteschlange,
Implementationsvarianten der verketteten Liste, amortisierte
Analyse [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap.
10.1.-10.2,17.1-17.3]

Abstrakte Datentypen

Wir erinnern uns¹³ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

¹³hoffentlich

Abstrakte Datentypen

Wir erinnern uns¹³ (Vorlesung Informatik I)

Ein *Stack* ist ein abstrakter Datentyp (ADT) mit Operationen

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

¹³hoffentlich

Abstrakte Datentypen

Wir erinnern uns¹³ (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**.

¹³hoffentlich

Abstrakte Datentypen

Wir erinnern uns¹³ (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**.

Abstrakte Datentypen

Wir erinnern uns¹³ (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**.

Abstrakte Datentypen

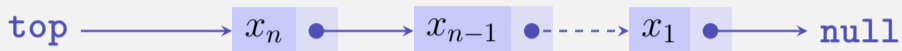
Wir erinnern uns¹³ (Vorlesung Informatik I)

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.

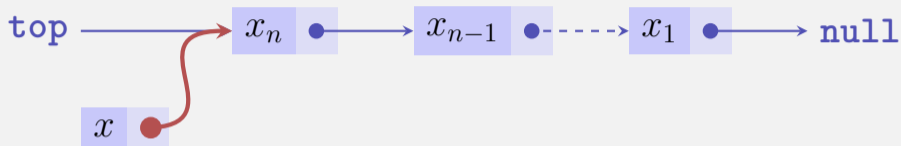
¹³hoffentlich

Implementation Push



`push(x, S):`

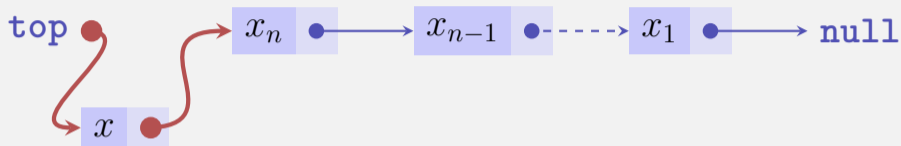
Implementation Push



`push`(x, S):

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

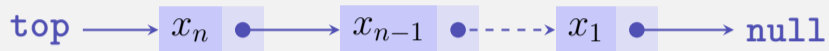
Implementation Push



`push`(x, S):

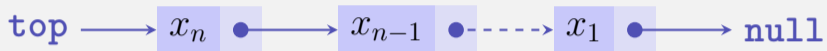
- 1 Erzeuge neues Listenelement mit x und Zeiger auf den Wert von `top`.
- 2 Setze `top` auf den Knoten mit x .

Implementation Pop



`pop(S)`:

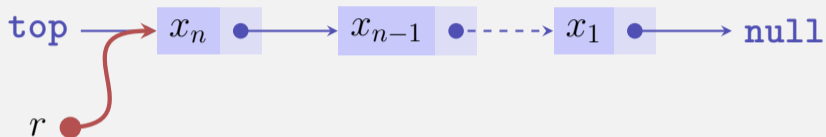
Implementation Pop



`pop(S)`:

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

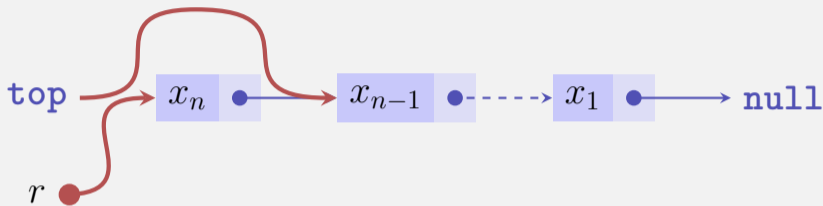
Implementation Pop



`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 Schlange zurück (**null** sonst.)

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

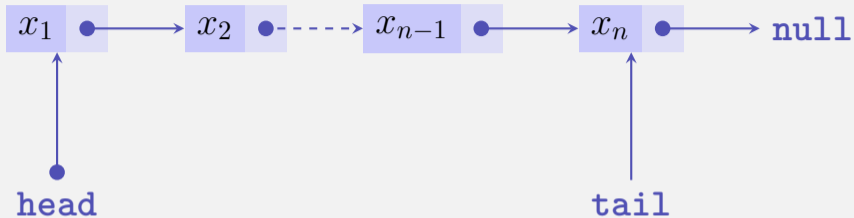
- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)
- **isEmpty**(Q): liefert **true** wenn Queue leer, sonst **false**.

Queue (Schlange / Warteschlange / Fifo)

Queue ist ein ADT mit folgenden Operationen:

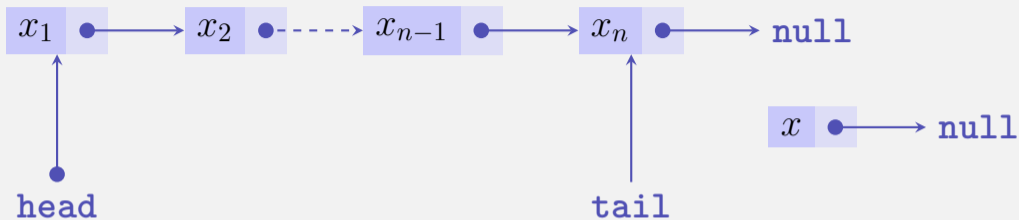
- **enqueue**(x, Q): fügt x am Ende der Schlange an.
- **dequeue**(Q): entfernt x vom Beginn der Schlange und gibt x zurück (**null** sonst.)
- **head**(Q): liefert das Objekt am Beginn der Schlange zurück (**null** sonst.)
- **isEmpty**(Q): liefert **true** wenn Queue leer, sonst **false**.
- **emptyQueue**():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

Implementation Queue



`enqueue(x, S):`

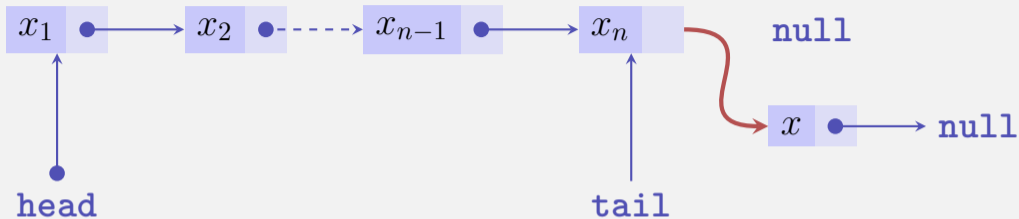
Implementation Queue



`enqueue`(x, S):

- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.

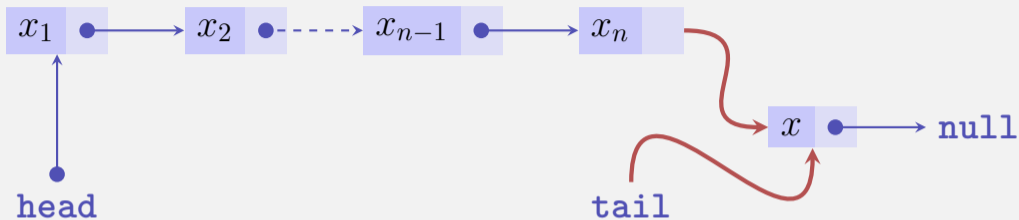
Implementation Queue



`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 .

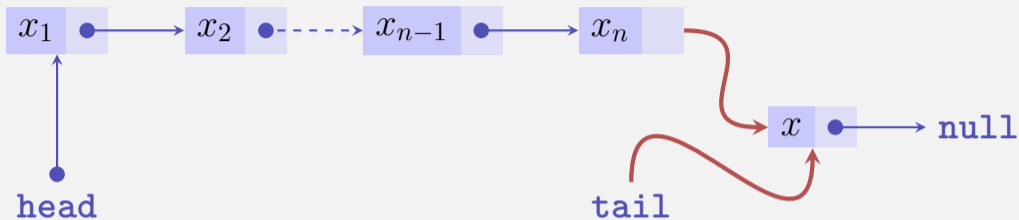
Implementation Queue



`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 .

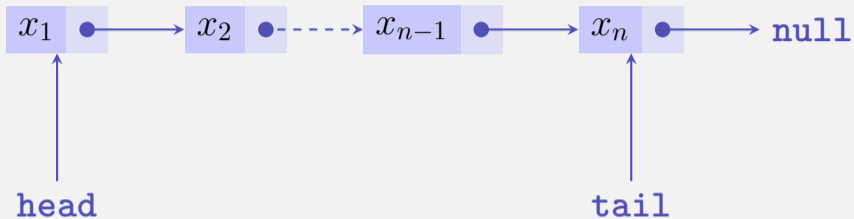
Implementation Queue



`enqueue(x, S):`

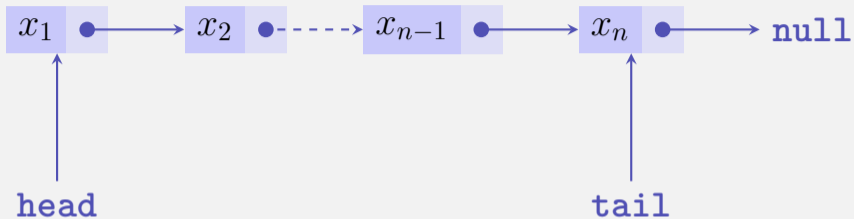
- 1 Erzeuge neues Listenelement mit x und Zeiger auf `null`.
- 2 Wenn `tail` \neq `null`, setze `tail.next` auf den Knoten mit x .
- 3 Setze `tail` auf den Knoten mit x .
- 4 Ist `head` = `null`, dann setze `head` auf `tail`.

Invarianten!



Mit dieser Implementation gilt

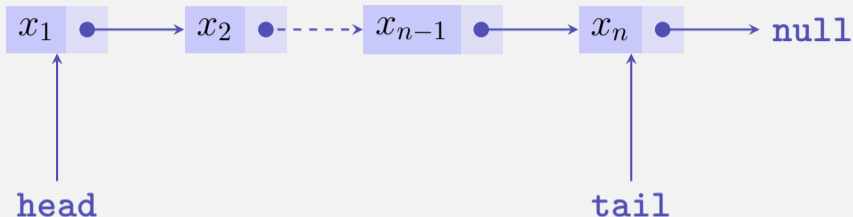
Invarianten!



Mit dieser Implementation gilt

- entweder `head = tail = null`,

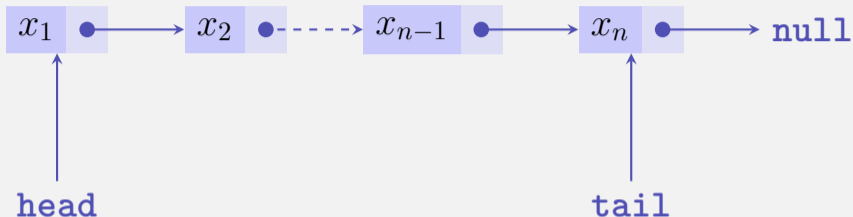
Invarianten!



Mit dieser Implementation gilt

- entweder `head = tail = null`,
- oder `head = tail \neq null` und `head.next = null`

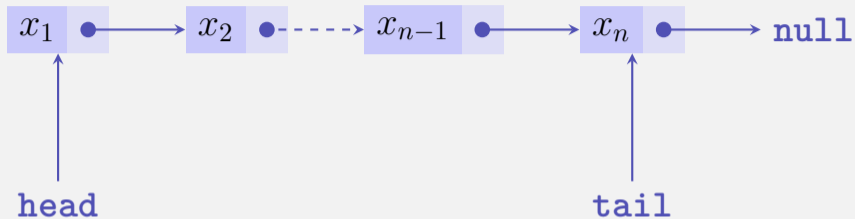
Invarianten!



Mit dieser Implementation gilt

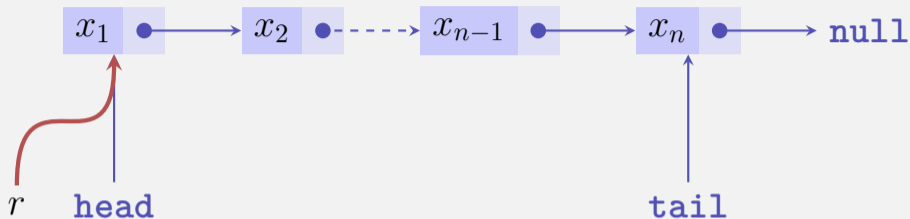
- entweder `head = tail = null`,
- oder `head = tail \neq null` und `head.next = null`
- oder `head \neq null` und `tail \neq null` und `head \neq tail` und `head.next \neq null`.

Implementation Queue



`dequeue(S)`:

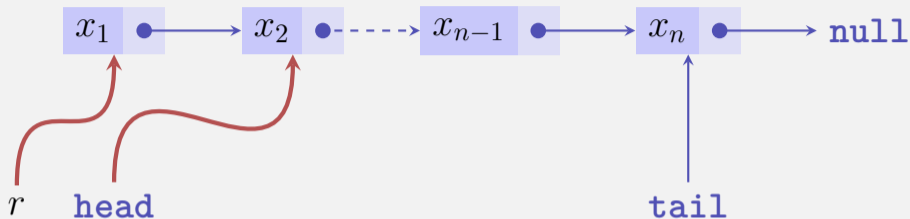
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, gib r zurück.

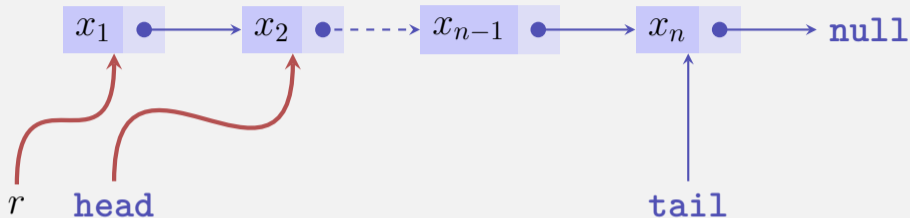
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, gib r zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.

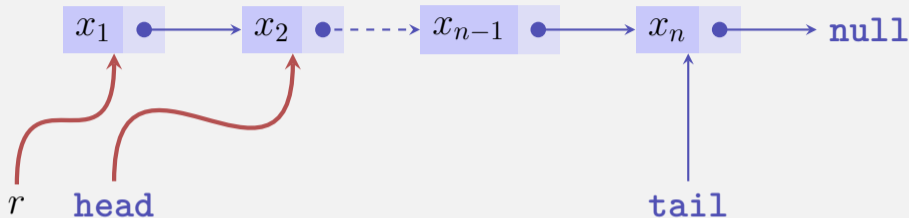
Implementation Queue



`dequeue(S)`:

- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, 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)`:

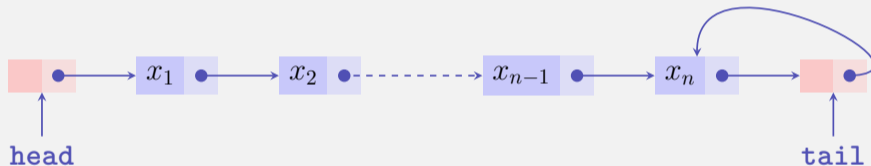
- 1 Merke Zeiger von `head` in r . Wenn $r = \text{null}$, gib r zurück.
- 2 Setze den Zeiger von `head` auf `head.next`.
- 3 Ist nun `head = null`, dann setze `tail` auf `null`.
- 4 Gib den Wert von r zurück.

Analyse

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

Implementationsvarianten verketteter Listen

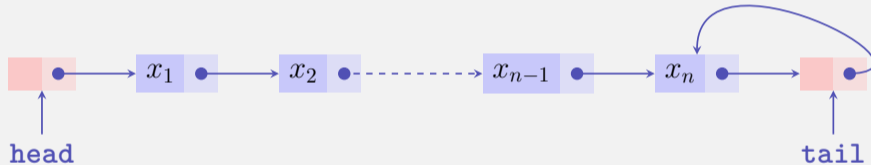
Liste mit Dummy-Elementen (Sentinels).



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.

Implementationsvarianten verketteter Listen

Doppelt verkettete Liste



Übersicht

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

(A) = Einfach verkettet

(B) = Einfach verkettet, mit Dummyelement

(C) = Einfach verkettet, mit einfach indirekter Elementadressierung

(D) = Doppelt verkettet

Prioritätswarteschlange (Priority Queue)

Priority Queue = Warteschlange mit Prioritäten.

Operationen

- `insert(x, p, Q)`: Füge Objekt x mit Priorität p ein.
- `extractMax(Q)`: Entferne Objekt x mit höchster Priorität und liefere es.

Implementation Prioritätswarteschlange

Mit einem Max-Heap!

Also

- `insert` in Zeit $\mathcal{O}(?)$ und
- `extractMax` in Zeit $\mathcal{O}(?)$.

Implementation Prioritätswarteschlange

Mit einem Max-Heap!

Also

- `insert` in Zeit $\mathcal{O}(\log n)$ und
- `extractMax` in Zeit $\mathcal{O}(?)$.

Implementation Prioritätswarteschlange

Mit einem Max-Heap!

Also

- `insert` in Zeit $\mathcal{O}(\log n)$ und
- `extractMax` in Zeit $\mathcal{O}(\log n)$.

Multistack

Multistack unterstützt neben den oben genannten Stackoperationen noch

`multiPop(s, 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 machen wir es besser?

Idee (Accounting)

Wir führen ein Kostenmodell ein:

- Aufruf von **push**: kostet 1 CHF und zusätzlich 1 CHF kommt aufs Bankkonto
- Aufruf von **pop**: kostet 1 CHF, wird durch Rückzahlung vom Bankkonto beglichen.

Kontostand wird niemals negativ. Also: maximale Kosten: Anzahl der **push** Operationen mal zwei.

Formalisierung

Bezeichne t_i die realen Kosten der Operation i . Potentialfunktion $\Phi_i \geq 0$ für den “Kontostand” nach i Operationen. $\Phi_i \geq \Phi_0 \forall i$.

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.

Beispiel Stack

Potentialfunktion $\Phi_i = \text{Anzahl Elemente auf dem Stack}$.

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

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.

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

$$\Rightarrow t_i = l + 1$$

Beispiel binärer Zähler

$$\dots 0 \underbrace{1111111}_l \text{ Einsen} + 1 = \dots 1 \underbrace{0000000}_l \text{ Nullen}$$

Potentialfunktion Φ_i : Anzahl der 1-Bits von x_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. 😊