# Datenstrukturen und Algorithmen

# Kapitel 21: Verschiedenes

Christian Scheideler

SS 2016

# Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherreallokation

# Union-Find Datenstruktur

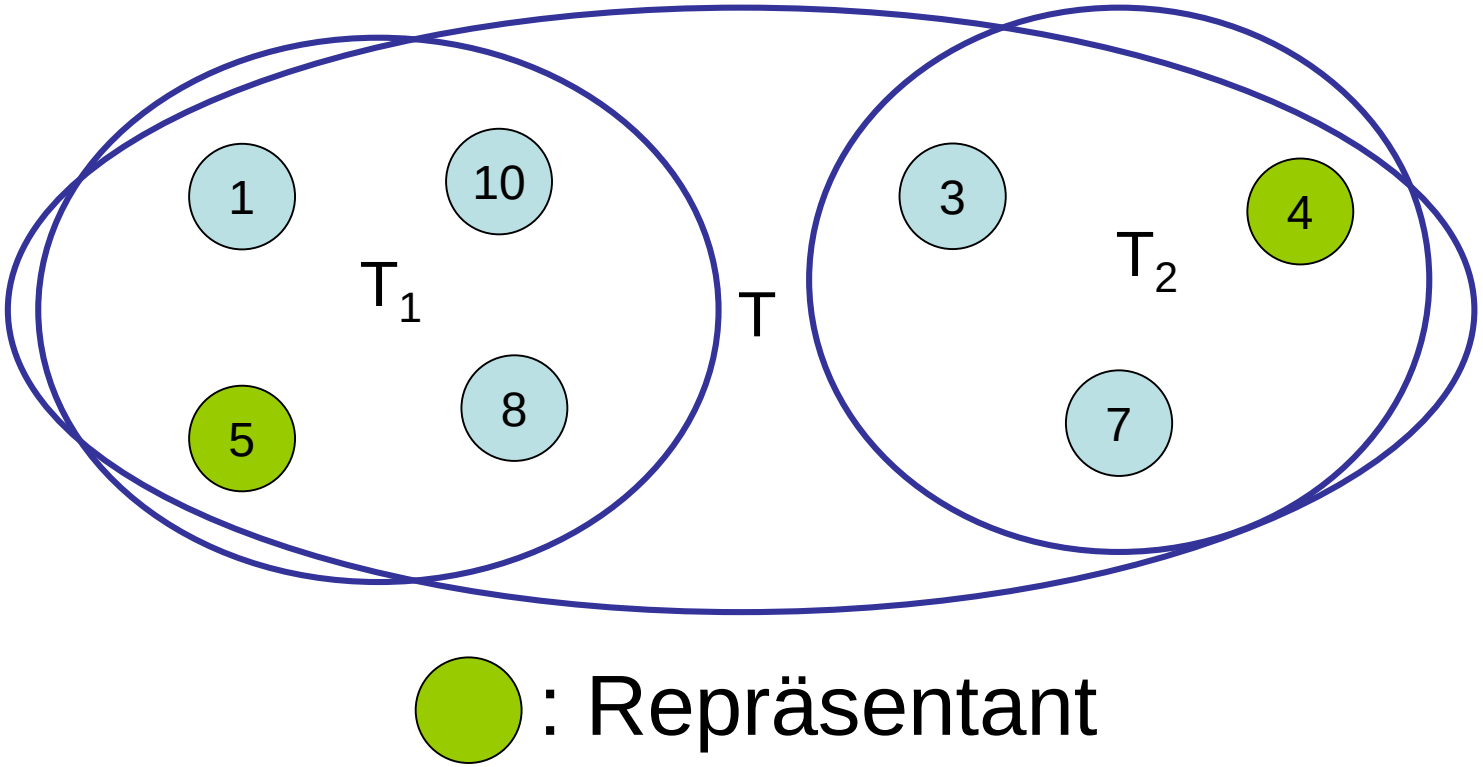
Gegeben: Menge von $n$ Teilmengen $T_1,\ldots,T_n$ die jeweils ein Element enthalten.

Operationen:

- Union($T_1$,$T_2$): vereinigt Elemente in $T_1$ und $T_2$ zu $T=T_1 \cup T_2$
- Find(x): gibt (eindeutigen) Repräsentanten der Teilmenge aus, zu der x gehört

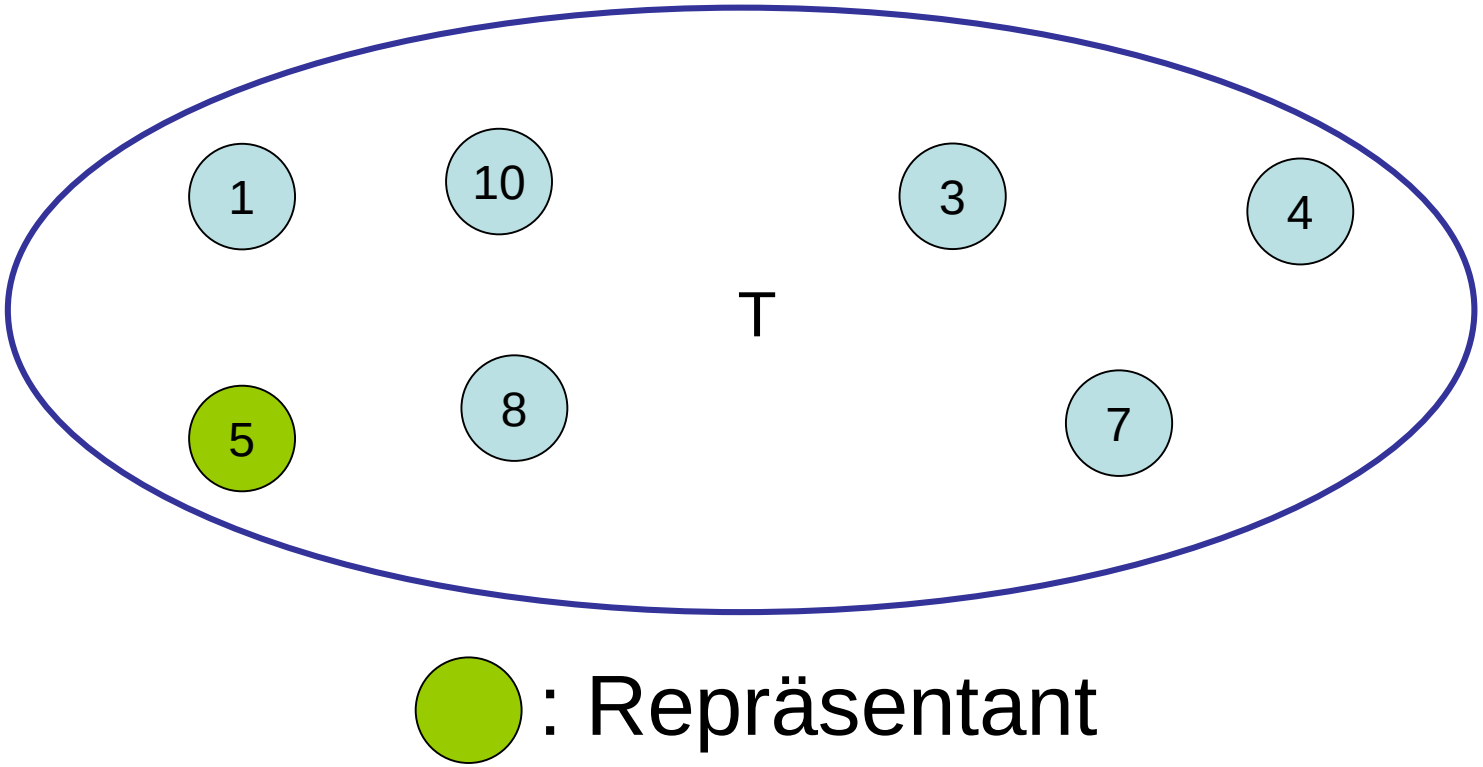
# Union-Find Datenstruktur

Union($T_1$,$T_2$):



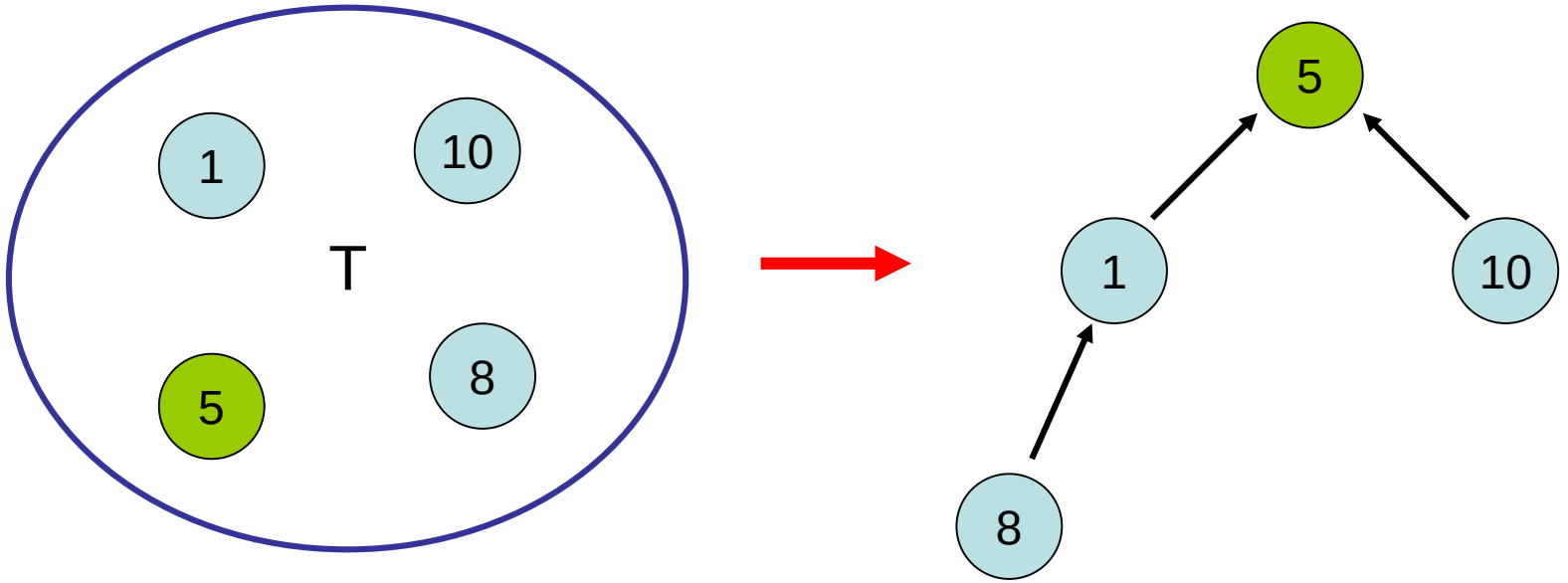
# Union-Find Datenstruktur

Find(10) liefert 5



# Union-Find Datenstruktur

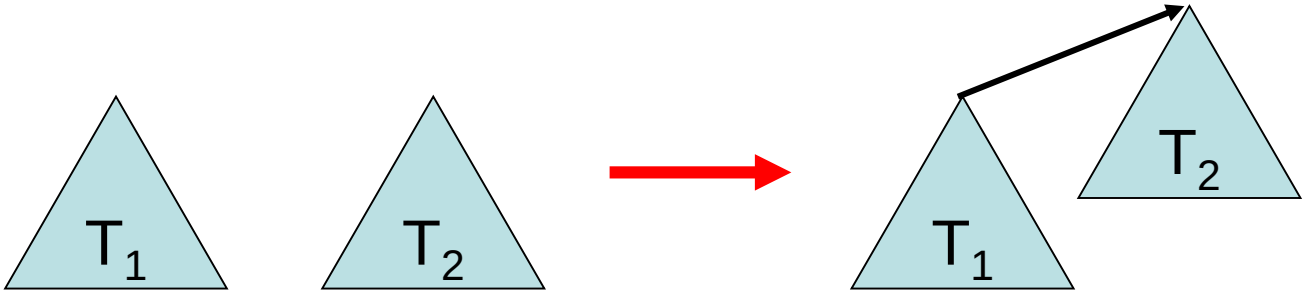
Idee: repräsentiere jede Menge T als gerichteten Baum mit Wurzel als Repräsentant



# Union-Find Datenstruktur

## Realisierung der Operationen:

- Union($T_1$,$T_2$):




- Find(x): Suche Wurzel des Baumes, in dem sich x befindet

# Union-Find Datenstruktur

Naïve Implementierung:

- Tiefe des Baums kann bis zu $n$ (bei $n$ Elementen) sein
- Zeit für Find: $\Theta(n)$ im worst case
- Zeit für Union: $O(1)$

# Union-Find Datenstruktur

Gewichtete Union-Operation: Mache die Wurzel des flacheren Baums zum Kind der Wurzel des tieferen Baums.

Lemma 21.1: Die Tiefe eines Baums ist höchstens $O(\log n)$.

Beweis:

- Die Tiefe von $T=T_1 \cup T_2$ erhöht sich nur dann, wenn $\text{Tiefe}(T_1)=\text{Tiefe}(T_2)$ ist
- $N(t)$: min. Anzahl Elemente in Baum der Tiefe $t$
- Es gilt $N(t)=2 \cdot N(t-1)$ und $N(0)=1$
- Also ist $N(\log n) = n$

# Union-Find Datenstruktur

Mit gewichteter Union-Operation:

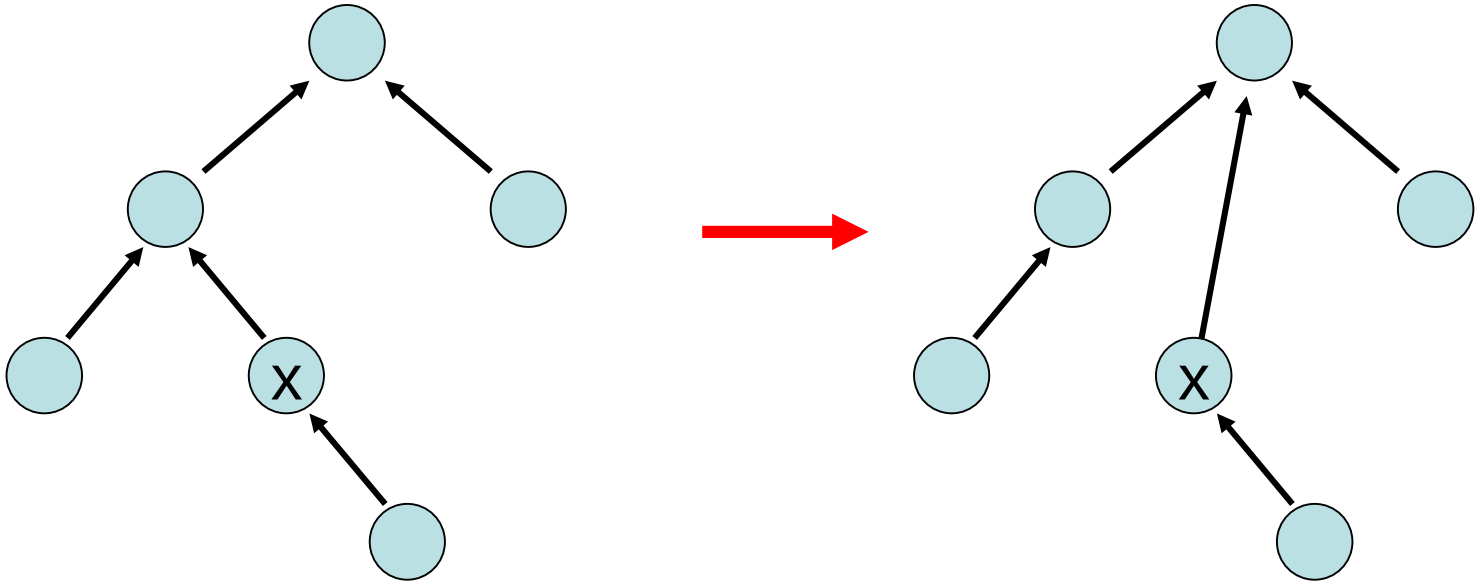
- Zeit für Find: $O(\log n)$
- Zeit für Union: $O(1)$

Es gilt auch: Tiefe eines Baums im worst-case $\Omega(\log n)$ (verwende Strategie, die Formel im Beweis von Lemma 21.1 folgt)

# Union-Find Datenstruktur

## Besser: gewichtetes Union mit Pfadkompression

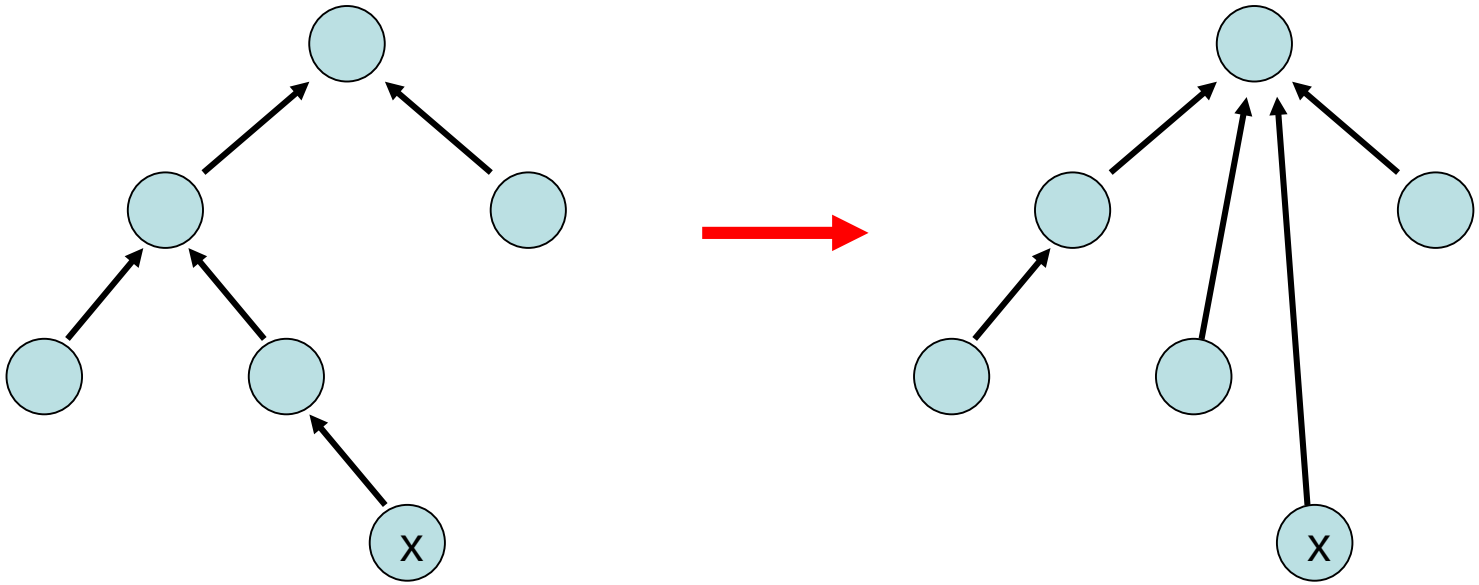
Pfadkompression bei jedem Find(x): alle Knoten von x zur Wurzel zeigen direkt auf Wurzel



# Union-Find Datenstruktur

## Besser: gewichtetes Union mit Pfadkompression

Pfadkompression bei jedem Find(x): alle Knoten von x zur Wurzel zeigen direkt auf Wurzel



# Union-Find Datenstruktur

Bemerkung: log* n ist definiert als

$$\log^* n = \min\{ i \geq 0 \mid \underbrace{\log \log \ \ldots \log}_{i\text{-mal}} n \leq 1\}$$

Beipiele:

- log* 0 = log* 1 = 0
- log* 2 = 1
- log* 4 = 2
- log* 16 = 3
- log* $2^{65536}$ = 5

# Union-Find Datenstruktur

Satz 21.2: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für Union und Find $O(\log^* n)$.

Beweis:

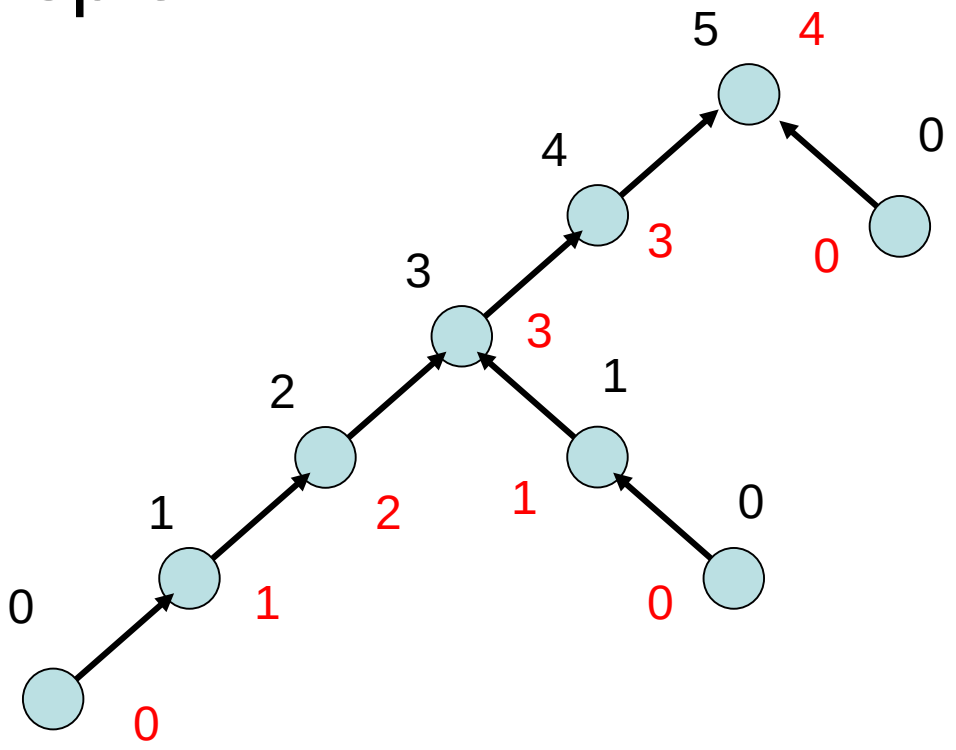
- T': endgültiger Baum, der durch die Folge der Unions ohne die Finds entstehen würde (also ohne Pfadkompression).

Ordne jedem Element x zwei Werte zu:

- rank(x) = Höhe des Unterbaums mit Wurzel x
- class(x) = i für das i mit $a_{i-1} < rank(x) \leq a_i$ wobei $a_{-1}=-1$, $a_0=0$ und $a_i = 2^{a_{i-1}}$ für alle $i>0$

# Union-Find Datenstruktur

Beispiel:




x: rank

x: class

# Union-Find Datenstruktur

Beweis (Fortsetzung):

- dist(x): Distanz von x zu nächstem Vorfahr y im tatsächlichen Union-Find-Baum T (mit Pfadkompression), so dass class(y)>class(x) ist, bzw. zur Wurzel y

Potenzialfunktion:

$$\Phi := c \sum_x \text{dist}(x)$$

für eine geeignete Konstante c>0.

# Union-Find Datenstruktur

Beobachtungen:

- Für den tatsächlichen Union-Find-Baum T seien x und y Knoten in T, y Vater von x. Dann ist class(x) ≤ class(y).
- Aufeinander folgende Find-Operationen durchlaufen (bis auf die letzte) verschiedene Kanten. Diese Kanten sind eine Teilfolge der Kanten in T' auf dem Pfad von x zur Wurzel.

# Union-Find Datenstruktur

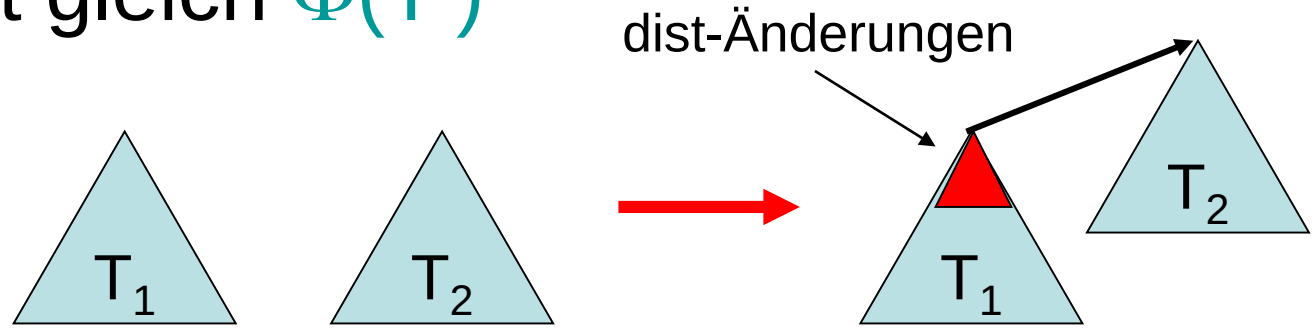
Amortisierte Kosten von Find($x_0$):

- $x_0 \rightarrow x_1 \rightarrow x_2 \ldots x_k$: Pfad von $x_0$ zur Wurzel in T'
- Es gibt höchstens log* n Kanten $(x_{i-1},x_i)$ mit $class(x_{i-1})<class(x_i)$
- Ist $class(x_{i-1})=class(x_i)$ und $i<k$, dann ist $dist(x_{i-1})$ vor der Find-Operation $\geq 2$ und nachher $=1$.
- Damit können die Kosten für alle Kanten $(x_{i-1},x_i)$ mit $class(x_{i-1})=class(x_i)$ aus der Potenzialverringerung bezahlt werden
- Amortisierte Kosten sind also O(log* n)

# Union-Find Datenstruktur

## Amortisierte Kosten von Union:

- dist-Änderungen über alle Unions bzgl. T' ist gleich $\Phi(T')$




- Potenzial von Baum T' mit n Knoten:

$$\Phi(T') \leq c \sum_{i=0}^{\log^* n} \sum_{x:\mathrm{rank}(x)=a_{i-1}+1}^{a_i} \mathrm{dist}(x)$$

# Union-Find Datenstruktur

$$\Phi(T') \leq c \sum_{i=0}^{\log^* n} \sum_{x:\mathrm{rank}(x)=a_{i-1}+1}^{a_i} \mathrm{dist}(x)$$
$$\leq c \sum_{i=0}^{\log^* n} (2 \cdot n / 2^{a_{i-1}+1}) a_i$$
$$\leq 2c \cdot n \sum_{i=0}^{\log^* n} a_i / 2^{a_{i-1}}$$
$$\leq 2c \cdot n \sum_{i=0}^{\log^* n} 1$$
$$= O(n \log^* n)$$

Die zweite Ungleichung gilt, da alle Unterbäume, deren Wurzel x rank(x)=j hat, disjunkt sind und jeweils $\geq 2^j$ Knoten enthalten.

# Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherreallokation

# Das Buddy System

Problem: Verwaltung freier Blöcke in einem gegebenen Adressraum {0,…,m-1} zur effizienten Allokation und Deallokation.


Vereinfachung:

- m ist eine Zweierpotenz
- nur Zweierpotenzen für allokierte Blockgrößen erlaubt

# Das Buddy System

$M \subseteq \{0,\ldots,m-1\}$: freier Adressraum

Operationen:

- Allocate(i): allokiert Block der Größe $2^i$, d.h. $M := M \setminus B$ für einen Block $B \subseteq M$ der Größe $2^i$
- Deallocate(B): gibt Block B wieder frei, d.h. $M := M \cup B$

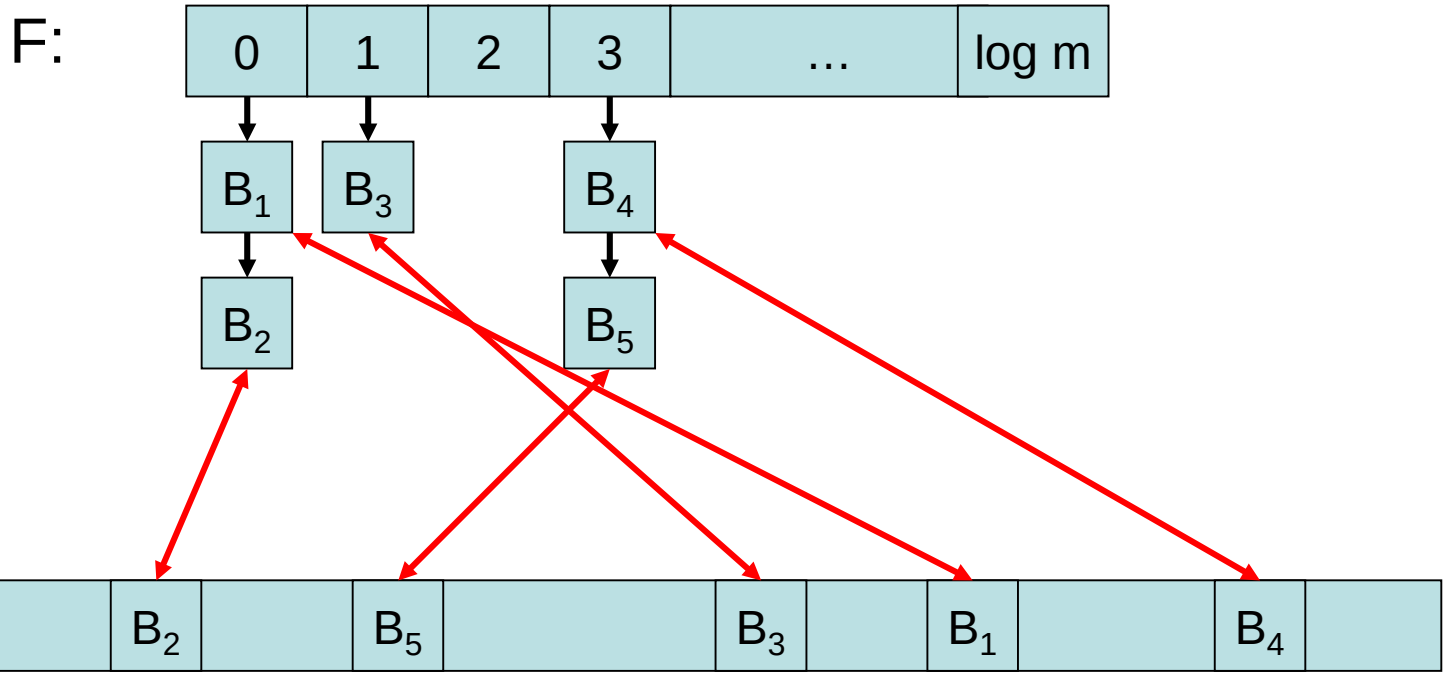
# Die Buddy Datenstruktur

## Datenstruktur:

- F: Feld von log m +1 (doppelt verketteten) Blocklisten F[0],…,F[log m]
- T: Hashtabelle mit Einträgen zu freien Blöcken. Jeder Eintrag enthält:
  - Startadresse addr(B) des Blocks B
  - Größe von Block B ($|B|=2^i$: speichere i)

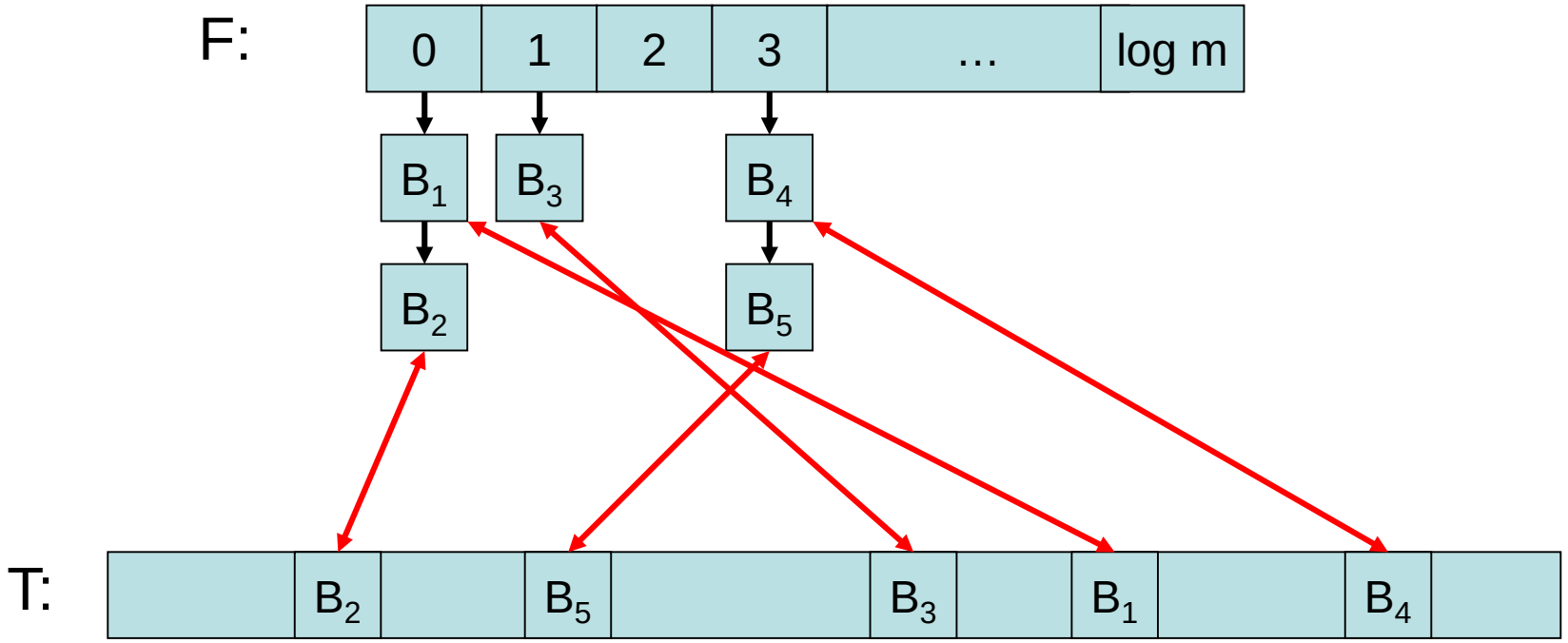
# Das Buddy System

F[i]: Liste von Blockgrößen $2^i$



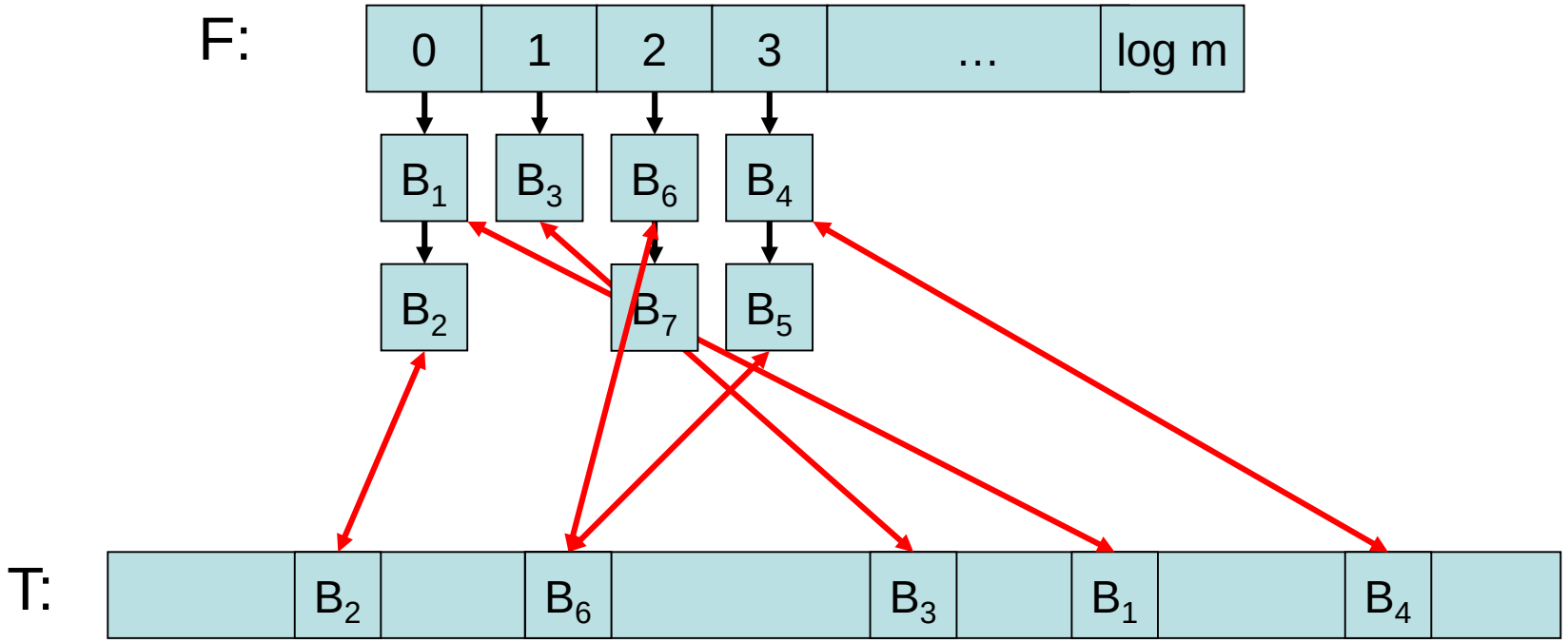
# Das Buddy System

Allocate(3):



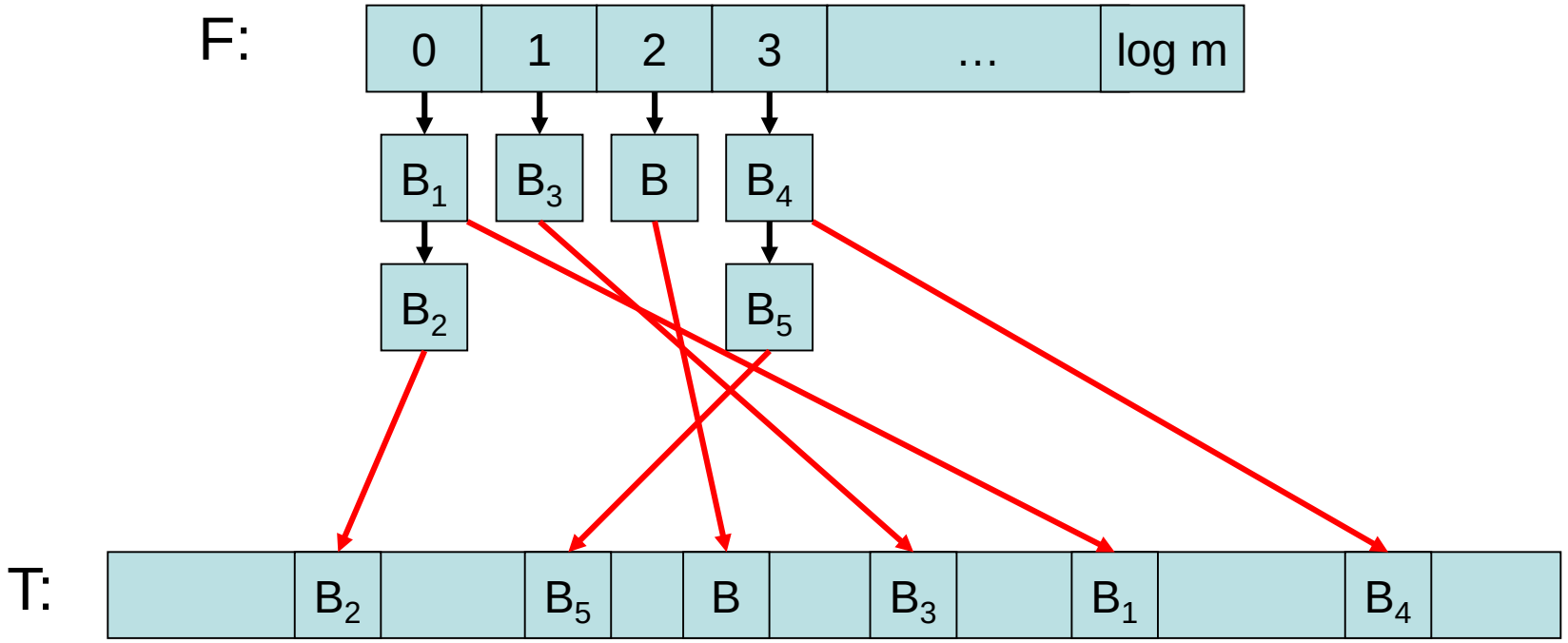
# Das Buddy System

Allocate(2):



# Das Buddy System

Deallocate(B): (Größe von B ist $2^2$)



# Das Buddy System

Problem: zunehmende Fragmentierung

Definition 21.3:

- Block B der Größe $2^i$ ist gültig: Startadresse von B ist 0 für die ersten i Bits
- Buddy von Block B der Größe $2^i$: Block B', für den $B \cup B'$ einen gültigen Block der Größe $2^{i+1}$ ergibt

Invariante: Für jeden freien Block B in Feld F ist der Buddy belegt.

# Das Buddy System

Bewahrung der Invariante bei Deallocate(B):

while Buddy(B) frei do
    B:=B ∪ Buddy(B)
    nimm Buddy(B) aus F und T raus
speichere B in F und T ab

Schnelle Bestimmung von Buddy(B):

- berechne Startadresse von B'=Buddy(B)
  (folgt direkt aus Startadresse von B)
- prüfe mittels T, ob B' existiert

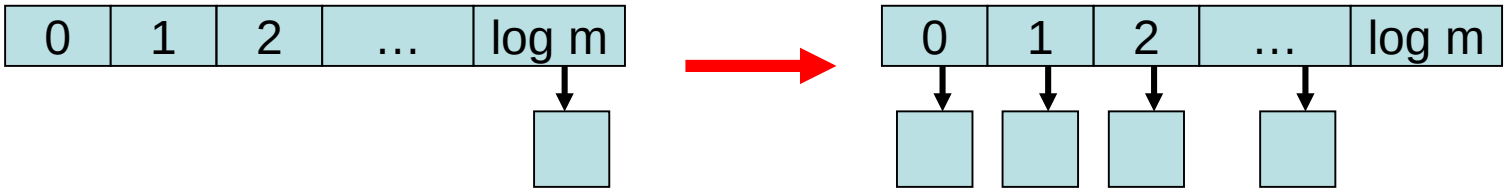
Schnelle Entfernung von Buddy(B):

- Bestimme über Hasheintrag Listenplatz von Buddy(B), um diesen schnell aus der Liste zu entfernen

# Das Buddy System

Probleme:

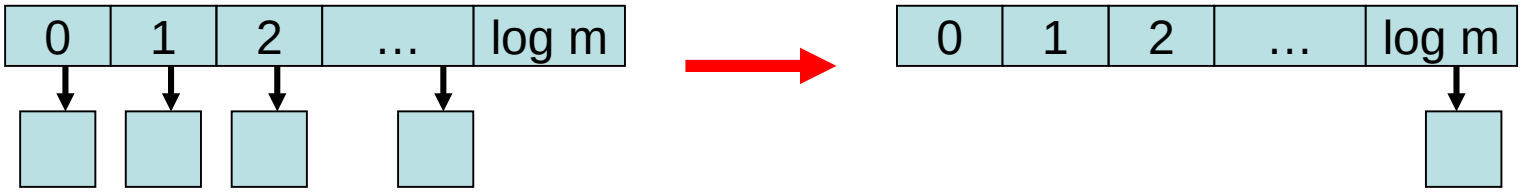
1. trotz Buddy-Ansatz keine garantiert niedrigen Obergrenzen für Fragmentierung
2. Allocate und Deallocate können $\Theta(\log m)$ viele Schritte laufen (wegen split- oder merge-Operationen)
   Beispiel: Allocate(0)



# Das Buddy System

Probleme:

1. trotz Buddy-Ansatz keine garantiert niedrigen Obergrenzen für Fragmentierung
2. Allocate und Deallocate können $\Theta(\log m)$ viele Schritte laufen (wegen split- oder merge-Operationen)
   Beispiel: Deallocate(0)



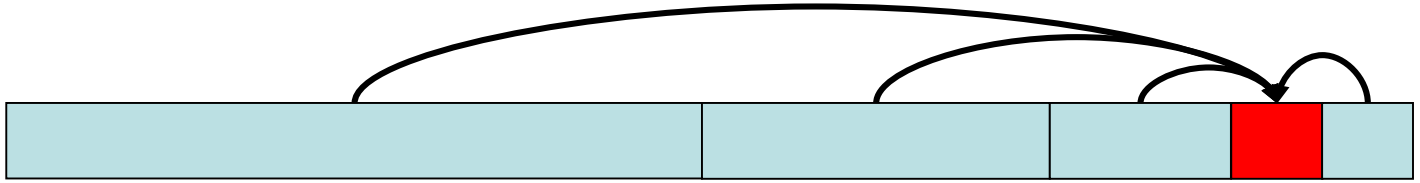
# Das Buddy System

1. Problem: Fragmentierung

Satz 21.4: Die Anzahl freier Blöcke ist höchstens
$O(\log m \cdot$ Anzahl allokierter Blöcke$)$.

Beweis:

- Freier Block nicht kombinierbar: allokierter Block dafür ein Zeuge



- Allokierter Block wird von höchstens $\log m$ freien Blöcken als Zeuge verwendet.

# Verbessertes Buddy System

2. Problem: Allocate und Deallocate brauchen Zeit $\Theta(\log m)$: kann effizient gelöst werden.

Idee: erlaube Blöcke, die freien Speicher der Form $2^k-2^i$ ($i<k$) angeben.

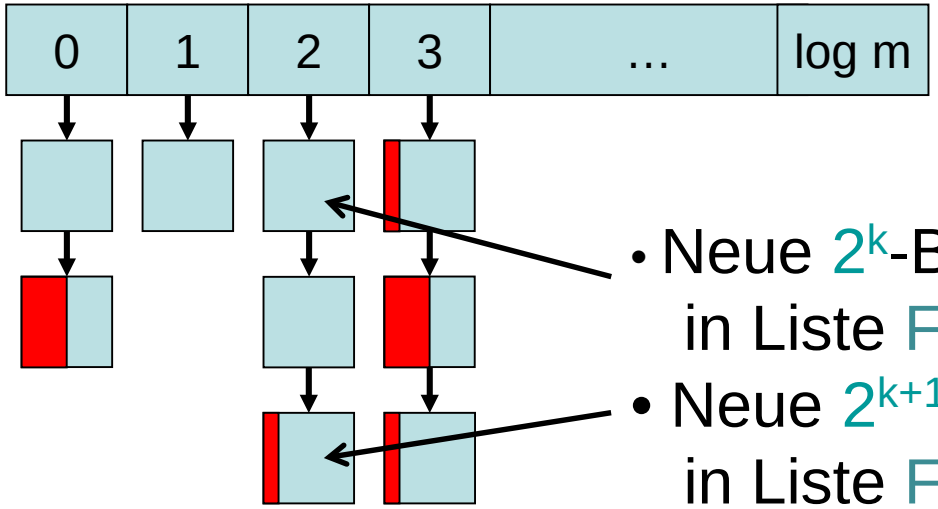

- $2^k-2^i$-Blöcke werden in F[k-1] gespeichert.
- Völlig freie $2^k$-Blöcke werden vorne in F[k] gespeichert.

# Verbessertes Buddy System

Beispiel:

Jedes F[i] merkt sich Anfang und Ende der Liste

F:

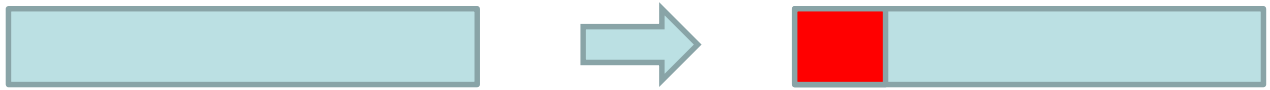



- Neue $2^k$-Blöcke vorne in Liste F[k] einfügen
- Neue $2^{k+1}$-$2^i$-Blöcke hinten in Liste F[k] einfügen

# Verbessertes Buddy System

Allocate(i): führe lazy splitting durch.

- Fall 1: $2^k$-Block B vorhanden, $k \geq i$.
  Schneide aus B vorderen $2^i$-Block raus, Rest von B wird $2^k$-$2^i$-Block.

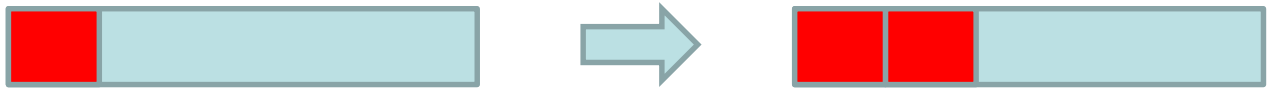


Laufzeit: O(1), falls Suche nach $2^k$-Block in O(1) Zeit machbar.

# Verbessertes Buddy System

Allocate(i): führe lazy splitting durch.

- Fall 2: $2^k$-$2^j$-Block B vorhanden, $k>i=j$.
  Schneide ersten gültigen $2^i$-Block aus B raus.
  Damit wird aus B ein $2^k$-$2^{i+1}$-Block.

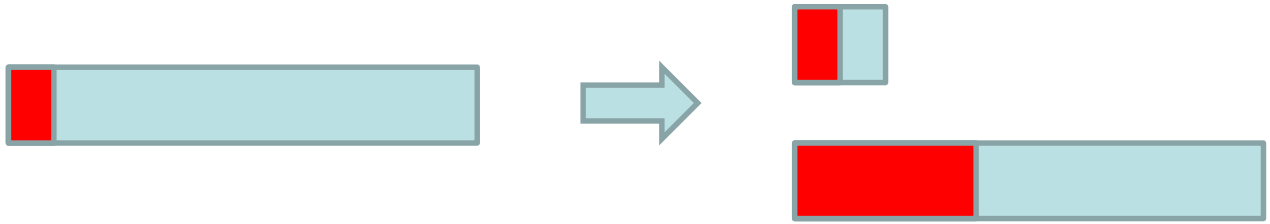


Laufzeit: O(1), falls Suche nach $2^k$-$2^j$-Block in O(1) Zeit machbar.

# Verbessertes Buddy System

Allocate(i): führe lazy splitting durch.

- Fall 3: $2^k$-$2^j$-Block B vorhanden, k>i>j.
  Schneide ersten gültigen $2^i$-Block aus B raus.
  Damit teilt sich B in $2^i$-$2^j$-Block und $2^k$-$2^{i+1}$-Block auf.

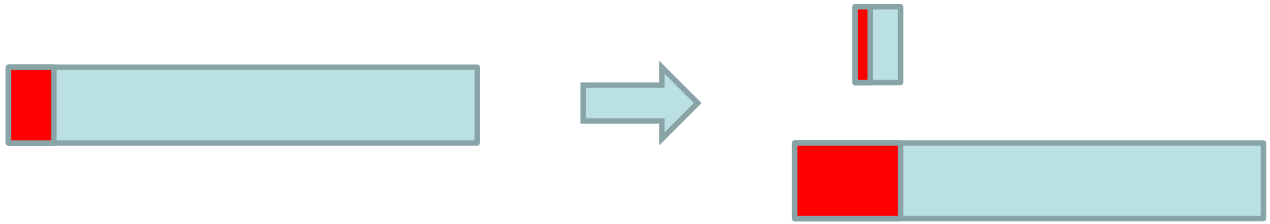


Laufzeit: O(1), falls Suche nach $2^k$-$2^j$-Block in O(1) Zeit machbar.

# Verbessertes Buddy System

Allocate(i): führe lazy splitting durch.

- Fall 4: $2^k$-$2^j$-Block B vorhanden, i<j.
  Schneide ersten gültigen $2^j$-Block aus B raus.
  Damit teilt sich B in $2^j$-$2^i$-Block und $2^k$-$2^{j+1}$-Block auf.

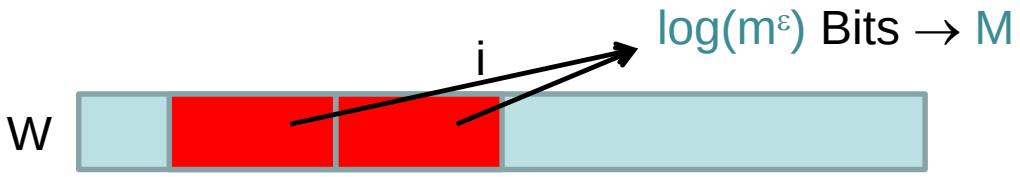


Laufzeit: O(1), falls Suche nach $2^k$-$2^j$-Block in O(1) Zeit machbar.

# Verbessertes Buddy System

O(1) Suchzeit nach passendem Block: Falls Worte der Größe $\log m + 1$ verfügbar, die mit Einheitskosten bearbeitet werden können, speichere zusätzlich zu $F$ ein Indexwort $W$. Setze Bit $i$ von $W$ auf 1 genau dann, wenn $F[i]$ ein Element enthält.

- Strategie 1:
  Berechne dann das erste gesetzte Bit $k \geq i$ bzw. $k > i$ in $W$ (was mit Pentium-Prozessoren in $O(1)$ Zeit machbar ist).
- Strategie 2 (falls Strategie 1 nicht in $O(1)$ machbar):
  Verwende Tabelle $M$ der Größe $m^\varepsilon$, die für jede Zahl $z \in \{0,\ldots,m^\varepsilon - 1\}$ das niedrigste gesetzte Bit in $z$ enthält. Damit maximal $1/\varepsilon$ Zeit, bis passender Index $k \geq i$ bzw. $k > i$ gefunden:



# Verbessertes Buddy System

Details zur Tabelle M:


Eintrag M(x) speichert Position des niedrigsten 1-Bits in x.

Berechnung des niedrigsten Bits in W ab Position i:

- Betrachte zuerst Bitfolge $(W_{i+k-1},\ldots,W_i)$ in W mit $k = \log(m^\varepsilon)$. Falls $(W_{i+k},\ldots,W_i)\neq 0$, dann greife auf $M(W_{i+k},\ldots,W_i)$ zu, um die Position des niedrigsten 1-Bits in $(W_{i+k},\ldots,W_i)$ zu bestimmen.
- Sonst fahre mit $(W_{i+2k-1},\ldots,W_{i+k})$ fort, usw., bis die Position des niedrigsten 1-Bits ermittelt wurde. (Ist das nicht der Fall, wird eine Fehlermeldung zurückgegeben.)

# Verbessertes Buddy System

Deallocate(B): führe lazy merging durch.

- Wiederhole, bis kein Fall mehr eintritt:
  - Fall 1: B hat freien Buddy B'=Buddy(B) in F:
    B:=B ∪ B'
  - Fall 2: B gehört zu freiem $2^k$-$2^i$-Block B' in F, Größe von B ist $2^i$:
    B:=B ∪ B'
- Speichere B in F (und T) ab

Laufzeit: amortisiert O(1).

# Verbessertes Buddy System

Lemma 21.5: Die amortisierte Laufzeit der Deallocate-Operation ist $O(1)$.

Beweis:

- Ein Merge wird nur dann auf $B$ und $B'$ angewandt, wenn $B \cup B'$ vorher in einem Allocate in $B$ und $B'$ geteilt worden ist.
- Gib jedem Block ein Potenzial, das angibt, dass er Ergebnis eines Splittings ist. Damit können Kosten des Mergens verrechnet werden.

# Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherreallokation

# Buddies mit Reallokation


Situation hier:

- Speicherreallokationen erlaubt, Kosten der Allokation proportional zur Speichergröße

Motivation: neu zugewiesener Speicherblock sollte keine Daten mehr enthalten, sonst Sicherheitsproblem!

- Allocate(i) hat (ohne Betrachtung der Kosten für die Reallokation) $2^i$ Zeitaufwand (Speicher wird überschrieben)
- Deallocate(i) hat keinen extra Zeitaufwand (Speicher wird lediglich freigegeben)

# Buddies mit Reallokation

Allocate(i):

- Annahme: zusammen mit neuem $2^i$-Block sind $(1-\varepsilon)m$ der Speicherzellen belegt.
- Suche gültigen $2^i$-Block B mit Belegung $<(1-\varepsilon)2^i$ (muss existieren!)
- Für alle Blöcke B’ in B:
  führe für B’ Allocate(log|B’|) auf (Rest-) Speicher ohne B aus
- Gib B zurück

Deallocate(B): wie im original Buddy-System

# Buddies mit Reallokation

Satz 21.6: Allocate(i) hat einen Zeitaufwand von höchstens $2^i/\varepsilon$.

Beweis:

- Allokation von Block B: Aufwand $2^i$
- Reallokation der belegten Blöcke B' in B: Aufwand $<(1-\varepsilon)2^i$
- Reallokation der dadurch verdrängten Blöcke $< (1-\varepsilon)^2\ 2^i$
- usw.
- Aufwand insgesamt max. $2^i \sum_{j\geq 0} (1-\varepsilon)^j = 2^i/\varepsilon$

# Buddies mit Reallokation

Bemerkungen:

- Satz 21.6 impliziert, dass der Aufwand für Allocate(i) $O(2^i)$ und damit asymptotisch optimal für das Reallokationsmodell ist, falls $\varepsilon$ konstant ist.
- Durch amortisierte Analyse kann man nachweisen, dass der Aufwand für Allocate(i) nur $O(1)$ ist, indem man nur diejenigen Teile des Blockes löscht bzw. umkopiert, die vorher beschrieben worden sind. (D.h. der Aufwand für einen Block hängt dann nicht mehr von seiner Größe sondern nur dem tatsächlich beschrie-benen Bereich ab.)