Effiziente Algorithmen und Datenstrukturen I

Kapitel 6: Verschiedenes

Christian Scheideler

WS 2008

Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherzugriffe
- DS zur Bandbreitenallokation

Union-Find Datenstruktur

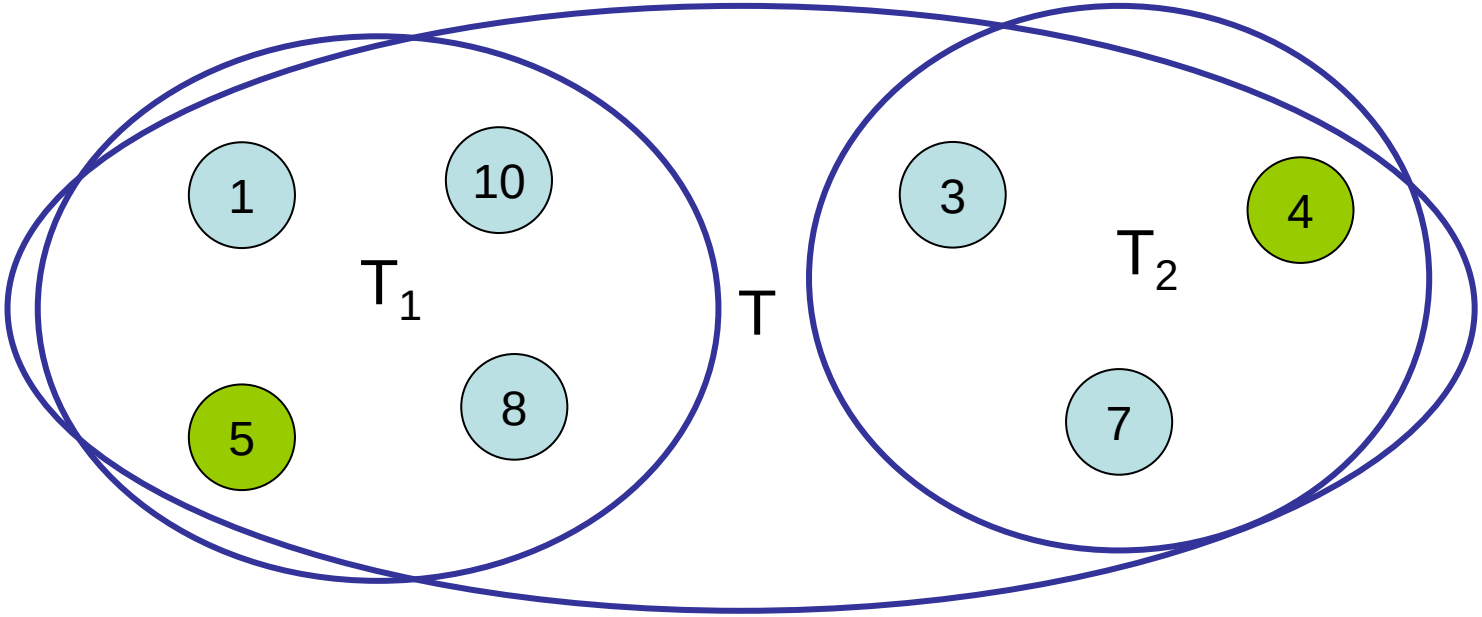
Gegeben: Menge von n Teilmengen $T_1,\ldots,T_n$ die jeweils ein Element enthalten.

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

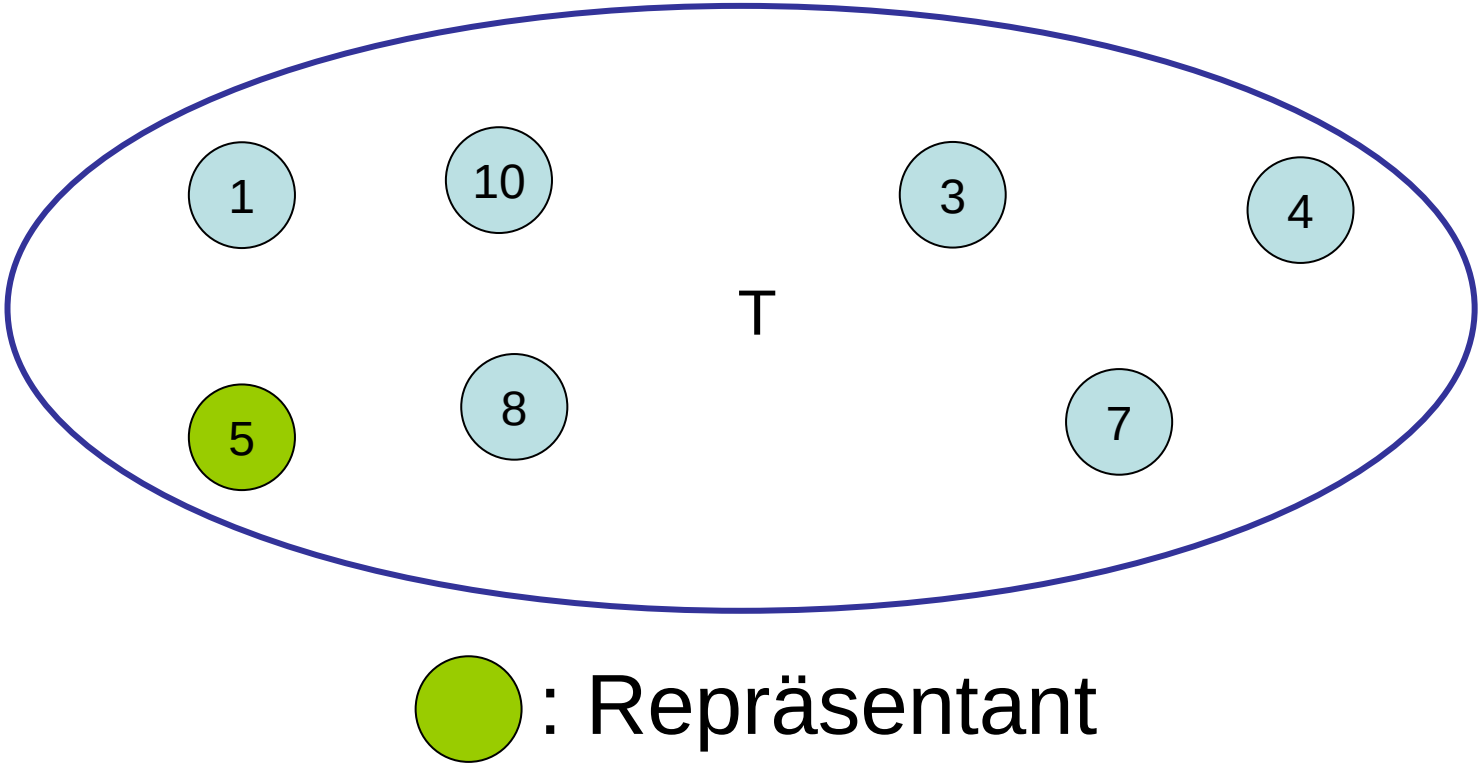
Union(T_1,T_2):



: Repräsentant

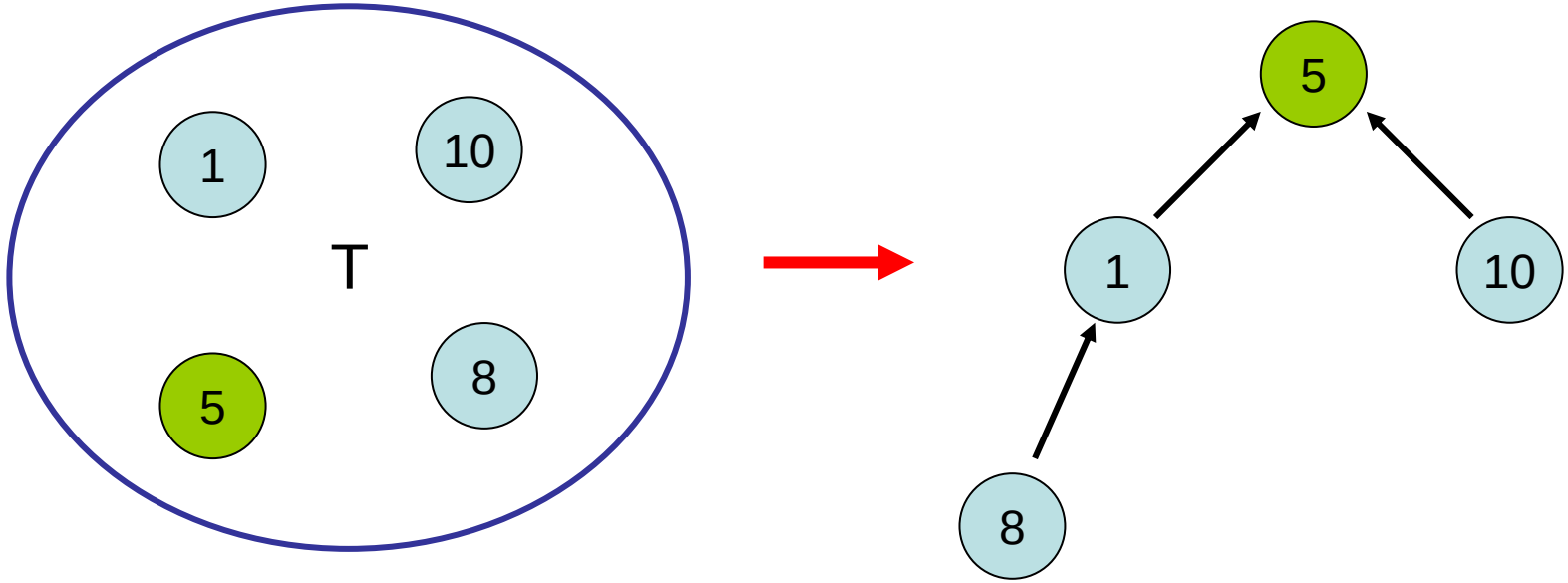
Union-Find Datenstruktur

Find(10) liefert 5



Union-Find Datenstruktur

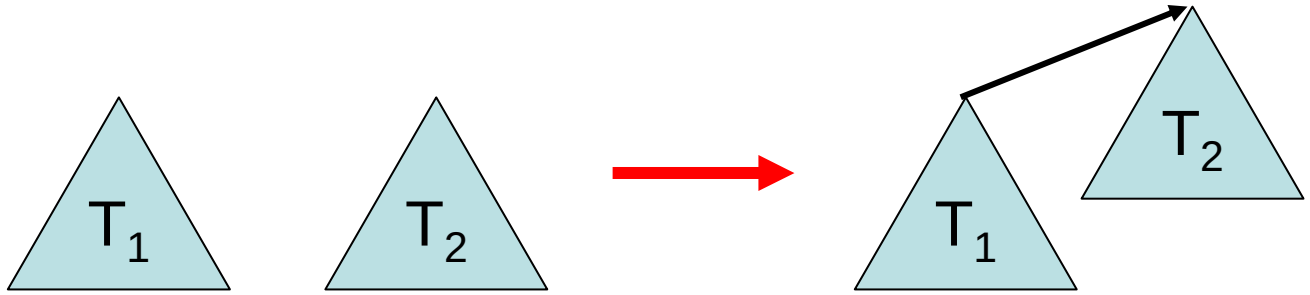
Idee: repräsentiere jede Menge T als gerich-teten Baum mit Wurzel als Repräsentant



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 6.1: 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:

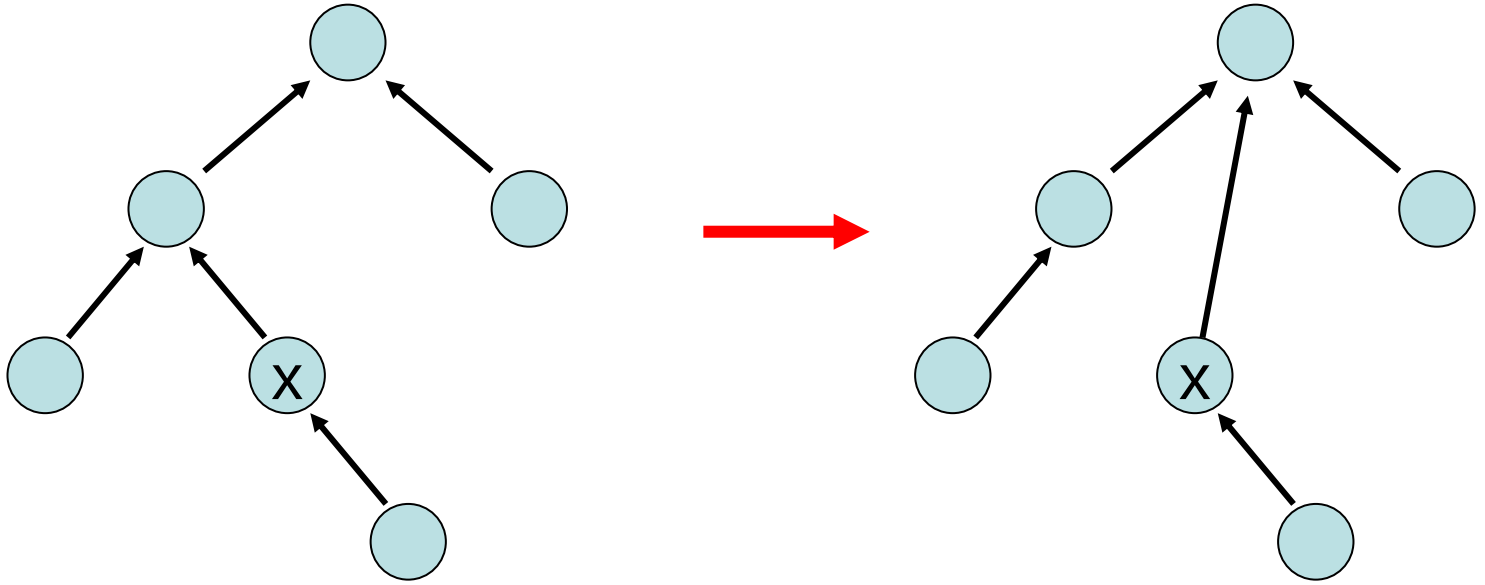
- Zeit für Find: $O(\log n)$
- Zeit für Union: $O(1)$

Es gilt auch: Tiefe eines Baums im worst-case $\Omega(\log n)$ (verwende Strategie, die Formel im Beweis von Lemma 6.1 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

Bemerkung: $\log^* n$ ist definiert als

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

Beipiele:

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

Union-Find Datenstruktur

Theorem 6.2: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für Union und Find $O(\log^* n)$.

Beweis:

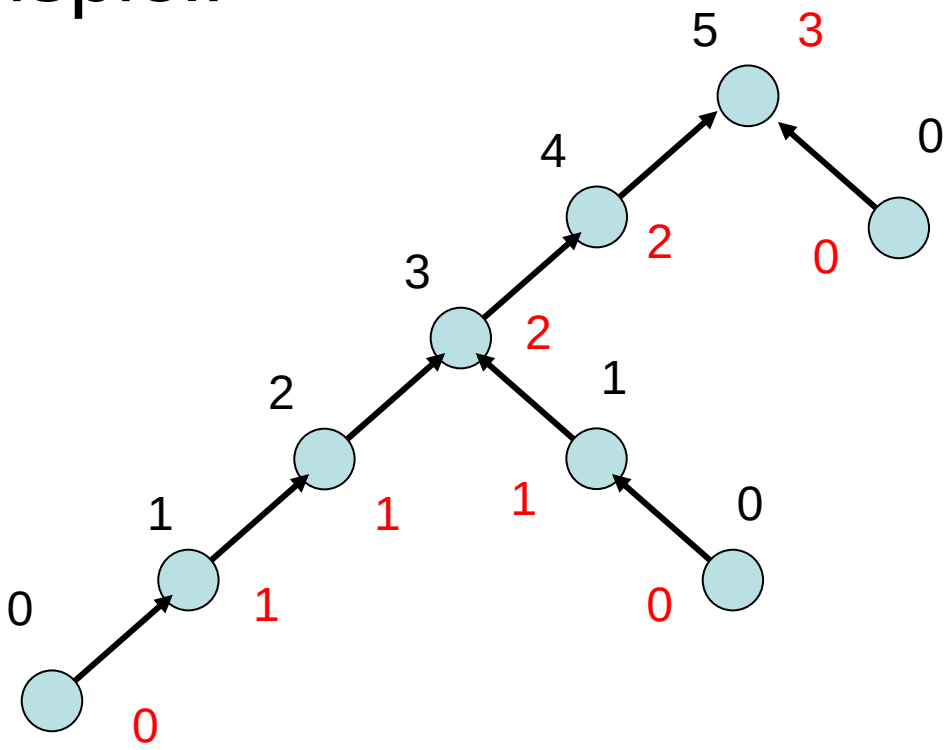
- T': endgültiger Baum, der durch die Folge der Unions ohne die Finds entstehen würde (also ohne Pfadkompression).

Ordne jedem Element x zwei Werte zu:

- rank(x) = Höhe des Unterbaums mit Wurzel x
- class(x) = i für das i mit $a_{i-1} < rank(x) \leqslant a_i$ 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 einem 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) $\leqslant$ 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

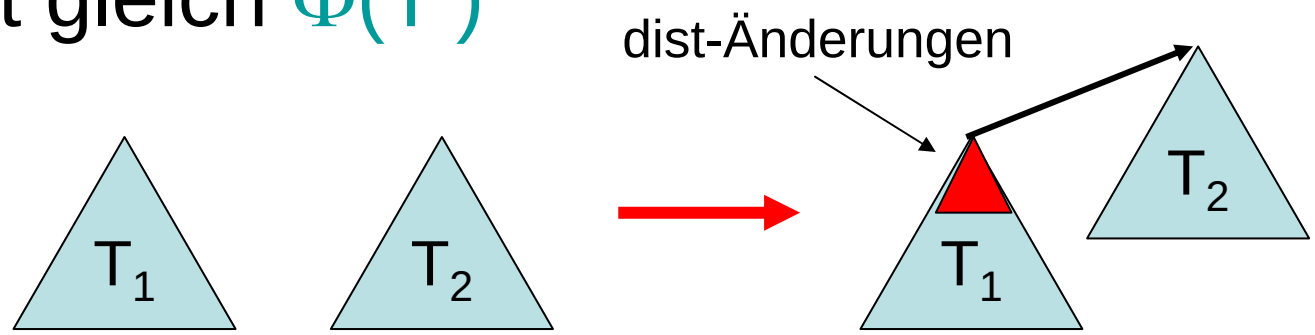
Amortisierte Kosten von Find:

- $x_0 \rightarrow x_1 \rightarrow x_2 \ldots x_k$: 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 $\geqslant 2$ 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:

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

Union-Find Datenstruktur

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

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

Union-Find Datenstruktur

Bessere obere Schranke mit $\alpha(k,n)$, $k \geqslant n$.

Betrachte die Ackermannfunktion $A(m,n)$ mit

- $A(0,n) = 2n$
- $A(m,0) = 2$
- $A(m+1,n+1) = A(m,A(m+1,n))$

Die Ackermannfunktion $A(n,n)$ steigt asymptotisch schneller als jede primitiv rekursive Funktion (d.h. auch $\log^* n$).

Union-Find Datenstruktur

Weiter wird gesetzt:

$$\alpha(k,n) = \min\{\, i \geqslant 1 \mid A(i, \lfloor k/n \rfloor) > \log n \,\}$$

Dann gilt: Der amortisierte Zeitbedarf für eine Folge von k Find- und $n-1$ Union-Operationen auf einer Menge von n Elementen ist $O(\alpha(k,n))$.

Es gibt auch eine entsprechende untere Schranke.

Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherzugriffe
- DS zur Bandbreitenallokation

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

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

Operationen:

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

Die Buddy Datenstruktur

Physikalischer Adressraum:

- F: Feld von $\log m + 1$ Blocklisten
  F[i], $i \geqslant 0$, speichert Anfangsadressen freier Blöcke der Größe 2^i
- Verwende das erste Byte von jedem allokierten Block der Größe 2^i zur Speicherung von $i+1$ (das reicht, da damit Blockgrößen bis zu $2^{2^8-2} = 2^{254}$ möglich).
  Freie Blöcke haben dieses Byte auf 0 gesetzt.

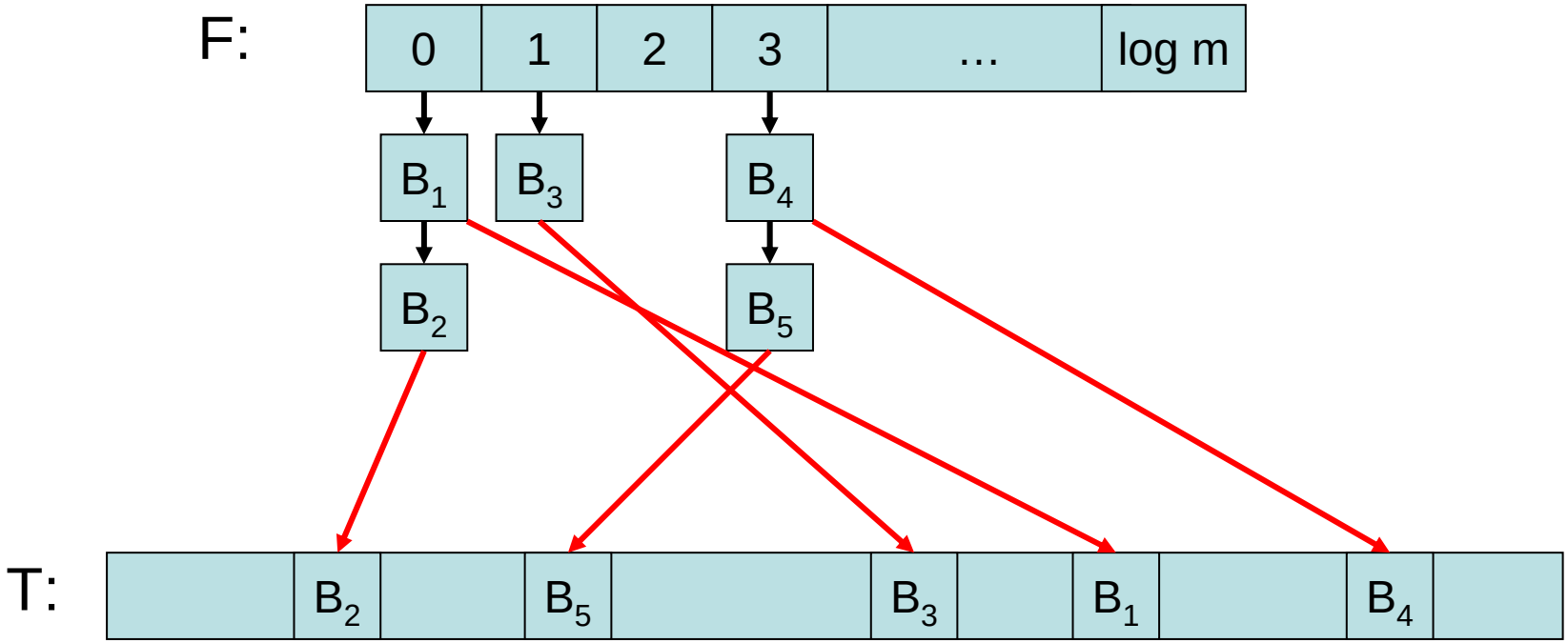
Die Buddy Datenstruktur

Virtueller Adressraum:

- F: Feld von log m +1 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)

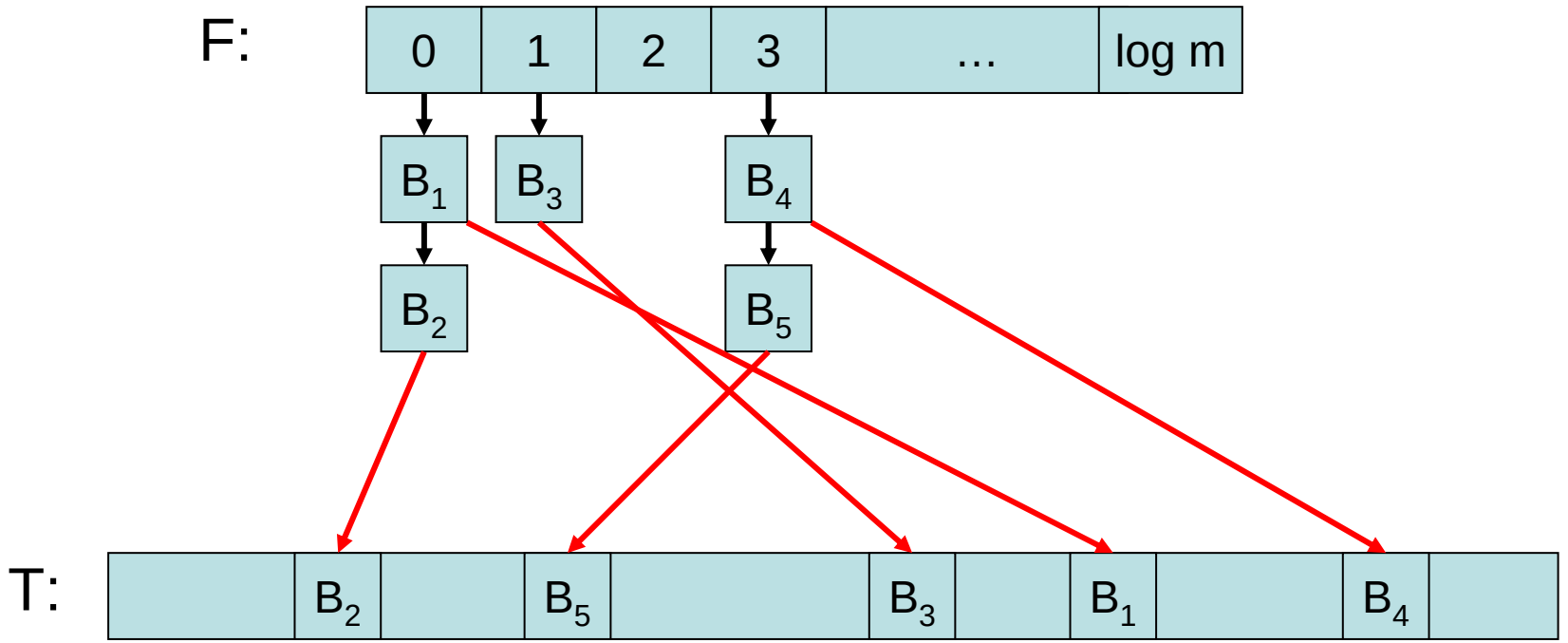
Das Buddy System

F[i]: Liste von Blockgrößen 2^i



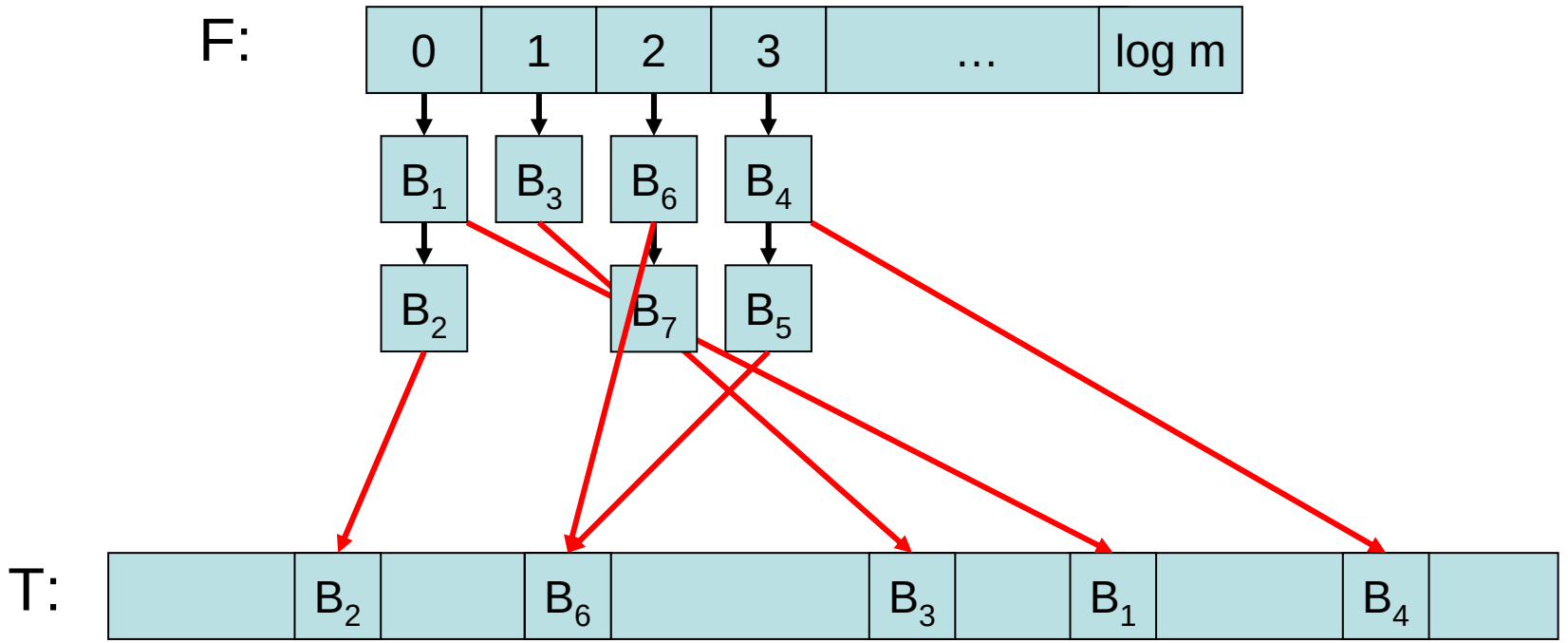
Das Buddy System

Allocate(3):



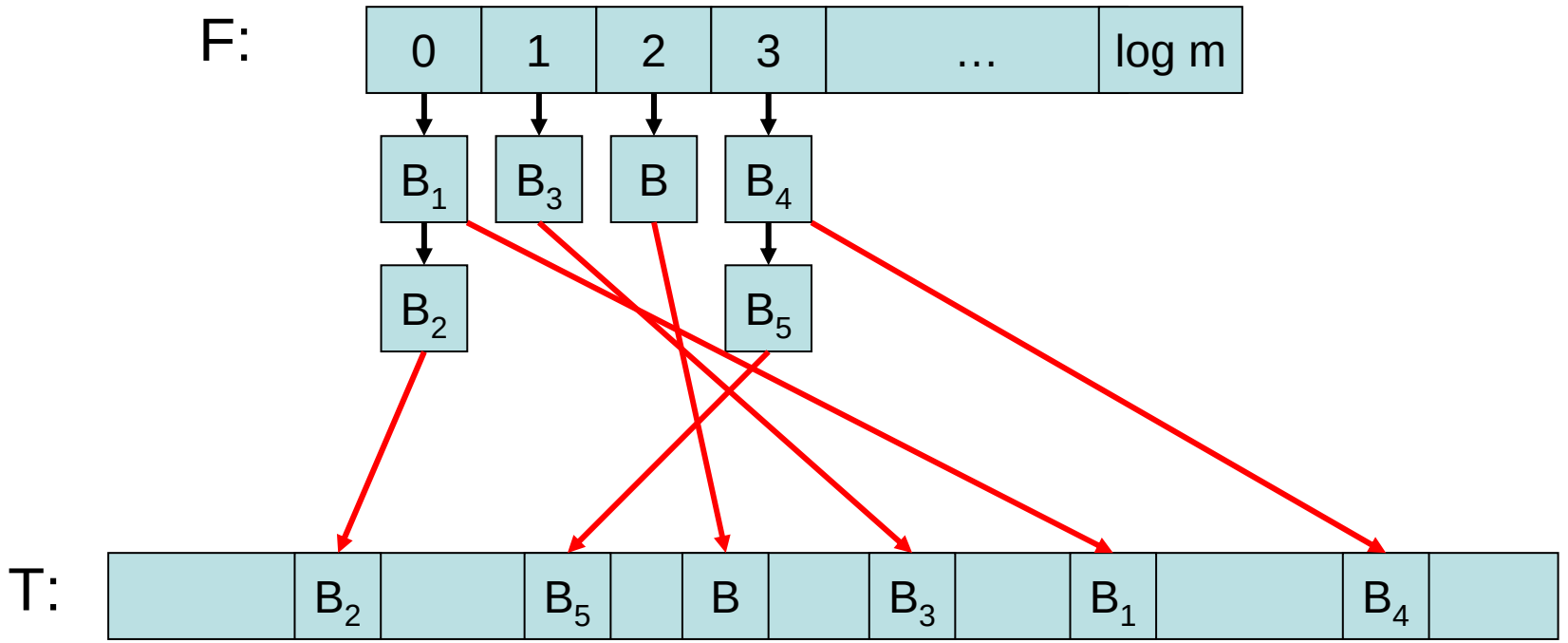
Das Buddy System

Allocate(2):



Das Buddy System

Deallocate(B): (Größe von B ist 2^2)



Das Buddy System

Problem: zunehmende Fragmentierung

Definition 6.3:

- 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)
speichere B in F und T ab

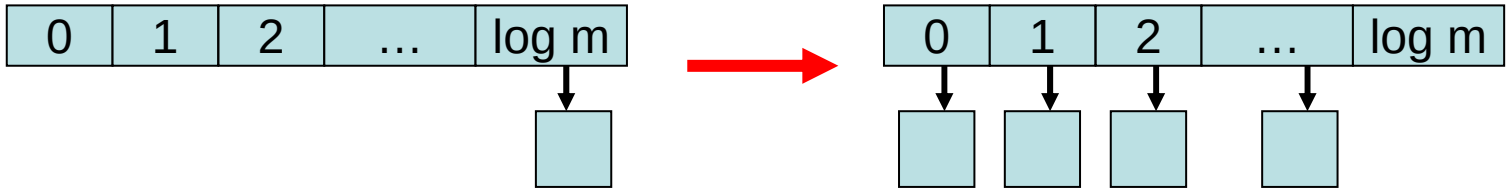
Schnelle Bestimmung von Buddy(B):

- berechne Startadresse von B'=Buddy(B) (folgt direkt aus Startadresse von B)
- prüfe mittels T, ob B' existiert (beim physikalischem Adressraum reicht es stattdessen, direkt auf das erste Byte von B' zuzugreifen)

Das Buddy System

Probleme:

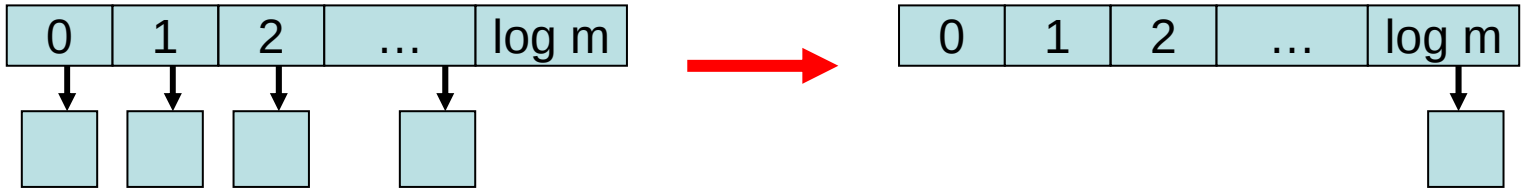
1. trotz Buddy-Ansatz keine garantiert niedrigen Obergrenzen für Fragmentierung
2. Allocate und Deallocate können $\Theta(\log m)$ viele Schritte laufen (wegen split- oder merge-Operationen)
   Beispiel: Allocate(0)



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)



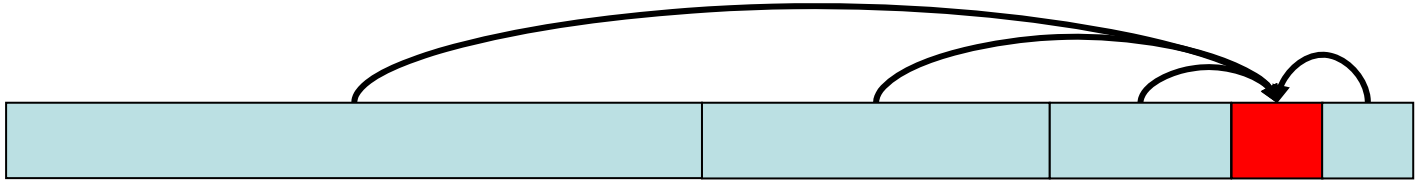
Das Buddy System

1. Problem: Fragmentierung

Theorem 6.4: 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.

Solche Blöcke werden in F[k-1] gespeichert.

Verbessertes Buddy System

Allocate(i): führe lazy splitting durch.

- Fall 1: 2^k-Block B vorhanden, $k \geqslant i$.
  Schneide aus B vorderen 2^i-Block raus, Rest von B wird 2^k-2^i-Block.
- Fall 2: nur 2^k-2^j-Block B vorhanden, $k>i$.
  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-Block bzw. 2^k-2^j-Block in O(1) Zeit machbar.

Verbessertes Buddy System

O(1) Suchzeit nach passendem Block:

- Strategie 1: Falls Worte der Größe $\log m + 1$ verfügbar, die mit Einheitskosten bearbeitet werden können, dann setze Bit i von Indexwort W auf 1 genau dann, wenn $F[i]$ ein Element enthält. Suche dann nach dem erstem gesetzten Bit $k \geqslant i$ bzw. $k > i$ (was mit Pentium-Prozessoren in $O(1)$ Zeit machbar ist).
- Strategie 2: Tabelle der Größe m^{ε}, 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 $k \geqslant i$ bzw. $k > i$ gefunden.

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: 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 Speicherzugriffe
- DS zur Bandbreitenallokation

Buddies mit Reallokation

Situation hier:

- Speicherreallokationen erlaubt
- Allocate(i) hat ohne Reallokation 2^i Zeit-aufwand (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:
  weise B’ Allocate(log|B’|) zu
- Gib B zurück

Deallocate(B): wie im original Buddy-System

Buddies mit Reallokation

Theorem 6.5: 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. $2^i \sum_{j\geqslant 0} (1-\varepsilon)^j = 2^i/\varepsilon$

Übersicht

- Union-Find Datenstruktur
- DS zur Speicherallokation
- DS für Speicherzugriffe
- DS zur Bandbreitenallokation

Bandbreitenallokation

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

Vereinfachung (wie bei Buddy-System):

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

Bandbreitenallokation

Situation hier: Reallokation von zugewiesenen Bandbreiten ist einfach machbar (O(1) Zeit pro Block)

Problem beim Buddy System: Reallokation teuer bzw. nicht möglich, da Speicherbereiche umkopiert werden müssen.

Bandbreitenallokation

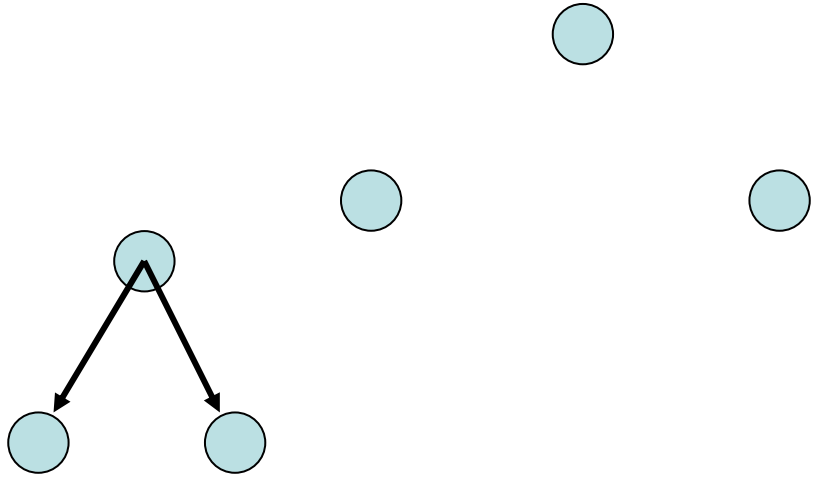
$M \subset \{0,\ldots,m-1\}$: freier Frequenzraum

Operationen:

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

Bandbreitenallokation

Datenstruktur: Binärbaum mit freien ( ) und belegten Blättern ( )



Bandbreitenallokation

Naiver Ansatz: Halte die belegten Blöcke sortiert nach Größe.

Das kann sehr viel Umorganisation verursachen.

Eine Strategie mit maximal 5 Umplatzierungen pro Allocate und Deallocate ist in “A Constant-Competitive Algorithm for Online OVSF Code Assignment” von F.Y.L. Chin, H.F. Ting und Y. Zhang zu finden.

Nächstes Kapitel

Wir fangen mit Graphalgorithmen an.